Coroutines and Tasks

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

Coroutines

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code (requires Python 3.7+) prints “hello”, waits 1 second, and then prints “world”:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine, asyncio provides three main mechanisms:

- The `asyncio.run()` function to run the top-level entry point "main()" function (see the above example.)
- Awaiting on a coroutine. The following snippet of code will print "hello" after waiting for 1 second, and then print "world" after waiting for *another* 2 seconds:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- The `asyncio.create_task()` function to run coroutines concurrently as asyncio `Tasks`.

Let's modify the above example and run two `say_after` coroutines *concurrently*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

Note that expected output now shows that the snippet runs 1 second faster than before:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of *awaitable* objects: **coroutines**, **Tasks**, and **Futures**.

Coroutines

Python coroutines are *awaitables* and therefore can be awaited from other coroutines:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Important: In this documentation the term “coroutine” can be used for two closely related concepts:

- a *coroutine function*: an `async def` function;
- a *coroutine object*: an object returned by calling a *coroutine function*.

asyncio also supports legacy generator-based coroutines.

Tasks

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:

```python
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A `Future` is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.

When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```python
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is `loop.run_in_executor()`.

Running an asyncio Program

`asyncio.`**run**(*coro*, *, *debug=False*)

Execute the coroutine *coro* and return the result.

This function runs the passed coroutine, taking care of managing the asyncio event loop, *finalizing asynchronous generators*, and closing the threadpool.

This function cannot be called when another asyncio event loop is running in the same thread.

If *debug* is `True`, the event loop will be run in debug mode.

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.

Example:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

New in version 3.7.

Changed in version 3.9: Updated to use `loop.shutdown_default_executor()`.

Note: The source code for `asyncio.run()` can be found in Lib/asyncio/runners.py.

Creating Tasks

`asyncio.`**`create_task`**(*coro*, *, *name=None*)

Wrap the *coro* coroutine into a `Task` and schedule its execution. Return the Task object.

If *name* is not `None`, it is set as the name of the task using `Task.set_name()`.

The task is executed in the loop returned by `get_running_loop()`, `RuntimeError` is raised if there is no running loop in current thread.

This function has been **added in Python 3.7**. Prior to Python 3.7, the low-level `asyncio.ensure_future()` function can be used instead:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
```

```
task = asyncio.ensure_future(coro())
...
```

New in version 3.7.

Changed in version 3.8: Added the `name` parameter.

Sleeping

coroutine `asyncio.`**`sleep`**(*`delay`*, *`result=None`*, *, *`loop=None`*)

Block for *delay* seconds.

If *result* is provided, it is returned to the caller when the coroutine completes.

`sleep()` always suspends the current task, allowing other tasks to run.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Example of coroutine displaying the current date every second for 5 seconds:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Running Tasks Concurrently

awaitable `asyncio.`**`gather`**(*`*aws`*, *`loop=None`*, *`return_exceptions=False`*)

Run awaitable objects in the *aws* sequence *concurrently*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in *aws*.

If *return_exceptions* is `False` (default), the first raised exception is immediately propagated to the task that awaits on `gather()`. Other awaitables in the *aws* sequence **won't be cancelled** and will continue to run.

If *return_exceptions* is `True`, exceptions are treated the same as successful results, and aggregated in the result list.

If `gather()` is *cancelled*, all submitted awaitables (that have not completed yet) are also *cancelled*.

If any Task or Future from the *aws* sequence is *cancelled*, it is treated as if it raised `CancelledError` – the `gather()` call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Example:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4)...
#     Task C: factorial(4) = 24
```

Note: If *return_exceptions* is False, cancelling gather() after it has been marked done won't cancel any submitted awaitables. For instance, gather can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching

an exception (raised by one of the awaitables) from gather won't cancel any other awaitables.

Changed in version 3.7: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

Shielding From Cancellation

awaitable `asyncio.`**`shield`**(*aw*, *, *loop=None*)

Protect an awaitable object from being `cancelled`.

If *aw* is a coroutine it is automatically scheduled as a Task.

The statement:

```
res = await shield(something())
```

is equivalent to:

```
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the "await" expression still raises a `CancelledError`.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a try/except clause, as follows:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Timeouts

coroutine `asyncio.`**`wait_for`**(*aw*, *timeout*, *, *loop=None*)

Wait for the *aw* awaitable to complete with a timeout.

If *aw* is a coroutine it is automatically scheduled as a Task.

timeout can either be `None` or a float or int number of seconds to wait for. If *timeout* is `None`, block until the future completes.

If a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`.

To avoid the task `cancellation`, wrap it in `shield()`.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*. If an exception happens during cancellation, it is propagated.

If the wait is cancelled, the future *aw* is also cancelled.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Example:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Changed in version 3.7: When *aw* is cancelled due to a timeout, `wait_for` waits for *aw* to be cancelled. Previously, it raised `asyncio.TimeoutError` immediately.

Waiting Primitives

coroutine `asyncio.`**`wait`**(*aws*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Run awaitable objects in the *aws* iterable concurrently and block until the condition specified by *return_when*.

The *aws* iterable must not be empty.

Returns two sets of Tasks/Futures: `(done, pending)`.

Usage:

```
done, pending = await asyncio.wait(aws)
```

timeout (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise `asyncio.TimeoutError`. Futures or Tasks that aren't done when the timeout occurs are simply returned in the second set.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
`FIRST_COMPLETED`	The function will return when any future finishes or is cancelled.
`FIRST_EXCEPTION`	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
`ALL_COMPLETED`	The function will return when all futures finish or are cancelled.

Unlike `wait_for()`, `wait()` does not cancel the futures when a timeout occurs.

Deprecated since version 3.8: If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task. Passing coroutines objects to `wait()` directly is deprecated as it leads to confusing behavior.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Note: `wait()` schedules coroutines as Tasks automatically and later returns those implicitly created Task objects in `(done, pending)` sets. Therefore the following code won't work as expected:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

Here is how the above snippet can be fixed:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
```

```
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

Deprecated since version 3.8, will be removed in version 3.11: Passing coroutine objects to `wait()` directly is deprecated.

`asyncio.`**`as_completed`**(*`aws`*, *, *`loop=None`*, *`timeout=None`*)

Run awaitable objects in the *aws* iterable concurrently. Return an iterator of coroutines. Each coroutine returned can be awaited to get the earliest next result from the iterable of the remaining awaitables.

Raises `asyncio.TimeoutError` if the timeout occurs before all Futures are done.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

Example:

```
for coro in as_completed(aws):
    earliest_result = await coro
    # ...
```

Running in Threads

coroutine `asyncio.`**`to_thread`**(*`func`*, */*, *`*args`*, *`**kwargs`*)

Asynchronously run function *func* in a separate thread.

Any *args and **kwargs supplied for this function are directly passed to *func*. Also, the current `contextvars.Context` is propagated, allowing context variables from the event loop thread to be accessed in the separate thread.

Return a coroutine that can be awaited to get the eventual result of *func*.

This coroutine function is primarily intended to be used for executing IO-bound functions/methods that would otherwise block the event loop if they were ran in the main thread. For example:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
```

```
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Directly calling *blocking_io()* in any coroutine would block the event loop for its duration, resulting in an additional 1 second of run time. Instead, by using *asyncio.to_thread()*, we can run it in a separate thread without blocking the event loop.

Note: Due to the GIL, *asyncio.to_thread()* can typically only be used to make IO-bound functions non-blocking. However, for extension modules that release the GIL or alternative Python implementations that don't have one, *asyncio.to_thread()* can also be used for CPU-bound functions.

New in version 3.9.

Scheduling From Other Threads

`asyncio.`**`run_coroutine_threadsafe`**(*coro*, *loop*)

Submit a coroutine to the given event loop. Thread-safe.

Return a `concurrent.futures.Future` to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

See the concurrency and multithreading section of the documentation.

Unlike other asyncio functions this function requires the *loop* argument to be passed explicitly.

New in version 3.5.1.

Introspection

`asyncio.`**`current_task`**(*`Loop=None`*)

Return the currently running `Task` instance, or `None` if no task is running.

If *loop* is `None` `get_running_loop()` is used to get the current loop.

New in version 3.7.

`asyncio.`**`all_tasks`**(*`Loop=None`*)

Return a set of not yet finished `Task` objects run by the loop.

If *loop* is `None`, `get_running_loop()` is used for getting current loop.

New in version 3.7.

Task Object

class `asyncio.`**`Task`**(*`coro`*, *, *`Loop=None`*, *`name=None`*)

A `Future-like` object that runs a Python coroutine. Not thread-safe.

Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level `asyncio.create_task()` function to create Tasks, or the low-level `loop.create_task()` or `ensure_future()` functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the `cancel()` method. Calling it will cause the Task to throw a `CancelledError` exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

`cancelled()` can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the `CancelledError` exception and was actually cancelled.

`asyncio.Task` inherits from `Future` all of its APIs except `Future.set_result()` and `Future.set_exception()`.

Tasks support the `contextvars` module. When a Task is created it copies the current context and later runs its coroutine in the copied context.

Changed in version 3.7: Added support for the `contextvars` module.

Changed in version 3.8: Added the `name` parameter.

Deprecated since version 3.8, will be removed in version 3.10: The *loop* parameter.

`cancel`(*msg=None*)

Request the Task to be cancelled.

This arranges for a `CancelledError` exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a `try` … … `except CancelledError` … `finally` block. Therefore, unlike `Future.cancel()`, `Task.cancel()` does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged.

Changed in version 3.9: Added the `msg` parameter.

The following example illustrates how coroutines can intercept the cancellation request:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
```

```
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

`cancelled`()

Return `True` if the Task is *cancelled*.

The Task is *cancelled* when the cancellation was requested with `cancel()` and the wrapped coroutine propagated the `CancelledError` exception thrown into it.

`done`()

Return `True` if the Task is *done*.

A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

`result`()

Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

If the Task has been *cancelled*, this method raises a `CancelledError` exception.

If the Task's result isn't yet available, this method raises a `InvalidStateError` exception.

`exception`()

Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

If the Task has been *cancelled*, this method raises a `CancelledError` exception.

If the Task isn't *done* yet, this method raises an `InvalidStateError` exception.

`add_done_callback`(*callback*, *, *context=None*)

Add a callback to be run when the Task is *done*.

This method should only be used in low-level callback-based code.

See the documentation of `Future.add_done_callback()` for more details.

`remove_done_callback`(*callback*)

Remove *callback* from the callbacks list.

This method should only be used in low-level callback-based code.

See the documentation of `Future.remove_done_callback()` for more details.

`get_stack`(*, *limit=None*)

Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional *limit* argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

`print_stack`(*, *limit=None*, *file=None*)

Print the stack or traceback for this Task.

This produces output similar to that of the traceback module for the frames retrieved by `get_stack()`.

The *limit* argument is passed to `get_stack()` directly.

The *file* argument is an I/O stream to which the output is written; by default output is written to `sys.stderr`.

get_coro()
Return the coroutine object wrapped by the `Task`.

New in version 3.8.

get_name()
Return the name of the Task.

If no name has been explicitly assigned to the Task, the default asyncio Task implementation generates a default name during instantiation.

New in version 3.8.

set_name(*value*)
Set the name of the Task.

The *value* argument can be any object, which is then converted to a string.

In the default Task implementation, the name will be visible in the `repr()` output of a task object.

New in version 3.8.

Generator-based Coroutines

Note: Support for generator-based coroutines is **deprecated** and is scheduled for removal in Python 3.10.

Generator-based coroutines predate async/await syntax. They are Python generators that use `yield from` expressions to await on Futures and other coroutines.

Generator-based coroutines should be decorated with `@asyncio.coroutine`, although this is not enforced.

`@asyncio.`**coroutine**
Decorator to mark generator-based coroutines.

This decorator enables legacy generator-based coroutines to be compatible with async/await code:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

This decorator should not be used for `async def` coroutines.

Deprecated since version 3.8, will be removed in version 3.10: Use `async def` instead.

`asyncio.`**`iscoroutine`**(*`obj`*)

Return `True` if *obj* is a coroutine object.

This method is different from `inspect.iscoroutine()` because it returns `True` for generator-based coroutines.

`asyncio.`**`iscoroutinefunction`**(*`func`*)

Return `True` if *func* is a coroutine function.

This method is different from `inspect.iscoroutinefunction()` because it returns `True` for generator-based coroutine functions decorated with `@coroutine`.