18.5.3. Tasks and coroutines

Source code: Lib/asyncio/tasks.py

Source code: Lib/asyncio/coroutines.py

18.5.3.1. Coroutines

Coroutines used with `asyncio` may be implemented using the `async def` statement, or by using generators. The `async def` type of coroutine was added in Python 3.5, and is recommended if there is no need to support older Python versions.

Generator-based coroutines should be decorated with `@asyncio.coroutine`, although this is not strictly enforced. The decorator enables compatibility with `async def` coroutines, and also serves as documentation. Generator-based coroutines use the `yield from` syntax introduced in **PEP 380**, instead of the original `yield` syntax.

The word "coroutine", like the word "generator", is used for two different (though related) concepts:

- The function that defines a coroutine (a function definition using `async def` or decorated with `@asyncio.coroutine`). If disambiguation is needed we will call this a *coroutine function* (`iscoroutinefunction()` returns `True`).
- The object obtained by calling a coroutine function. This object represents a computation or an I/O operation (usually a combination) that will complete eventually. If disambiguation is needed we will call it a *coroutine object* (`iscoroutine()` returns `True`).

Things a coroutine can do:

- `result = await future` or `result = yield from future` – suspends the coroutine until the future is done, then returns the future's result, or raises an exception, which will be propagated. (If the future is cancelled, it will raise a `CancelledError` exception.) Note that tasks are futures, and everything said about futures also applies to tasks.
- `result = await coroutine` or `result = yield from coroutine` – wait for another coroutine to produce a result (or raise an exception, which will be propagated). The `coroutine` expression must be a *call* to another coroutine.
- `return expression` – produce a result to the coroutine that is waiting for this one using `await` or `yield from`.
- `raise exception` – raise an exception in the coroutine that is waiting for this one using `await` or `yield from`.

Calling a coroutine does not start its code running – the coroutine object returned by the call doesn't do anything until you schedule its execution. There are two basic ways to start it running: call `await coroutine` or `yield from coroutine` from another coroutine (assuming the other coroutine is already running!), or schedule its execution using the `ensure_future()` function or the `AbstractEventLoop.create_task()` method.

Coroutines (and tasks) can only run when the event loop is running.

`@asyncio.`**`coroutine`**

Decorator to mark generator-based coroutines. This enables the generator use `yield from` to call `async def` coroutines, and also enables the generator to be called by `async def` coroutines, for instance using an `await` expression.

There is no need to decorate `async def` coroutines themselves.

If the generator is not yielded from before it is destroyed, an error message is logged. See Detect coroutines never scheduled.

Note: In this documentation, some methods are documented as coroutines, even if they are plain Python functions returning a `Future`. This is intentional to have a freedom of tweaking the implementation of these functions in the future. If such a function is needed to be used in a callback-style code, wrap its result with `ensure_future()`.

18.5.3.1.1. Example: Hello World coroutine

Example of coroutine displaying `"Hello World"`:

```
import asyncio

async def hello_world():
    print("Hello World!")

loop = asyncio.get_event_loop()
# Blocking call which returns when the hello_world() coroutine is done
loop.run_until_complete(hello_world())
loop.close()
```

See also: The Hello World with call_soon() example uses the `AbstractEventLoop.call_soon()` method to schedule a callback.

18.5.3.1.2. Example: Coroutine displaying the current date

Example of coroutine displaying the current date every second during 5 seconds using the `sleep()` function:

```
import asyncio
import datetime

async def display_date(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)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is dor
loop.run_until_complete(display_date(loop))
loop.close()
```

See also: The display the current date with call_later() example uses a callback with the `AbstractEventLoop.call_later()` method.

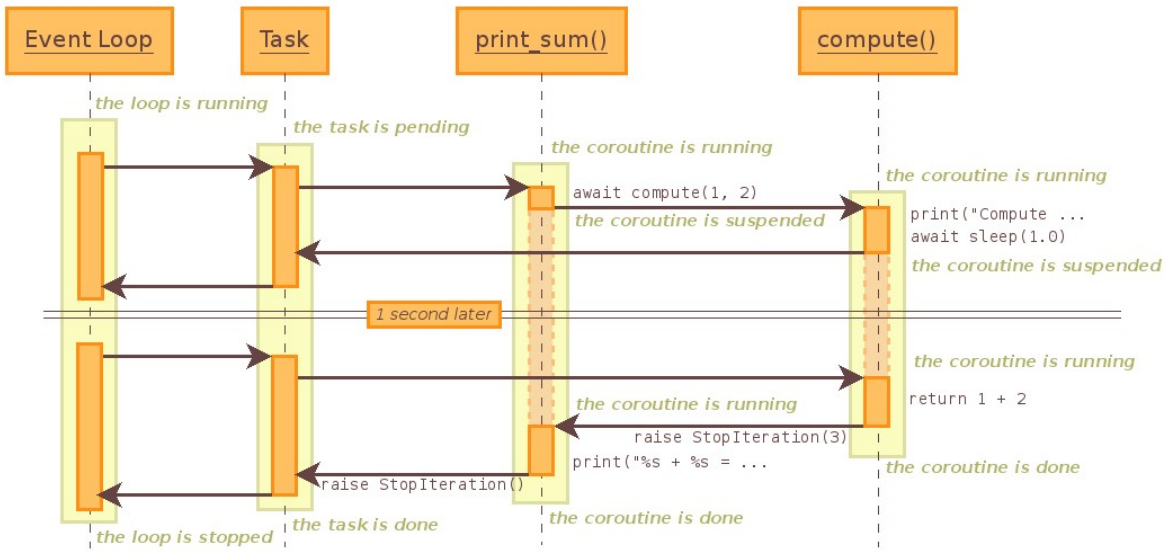
18.5.3.1.3. Example: Chain coroutines

Example chaining coroutines:

```
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

`compute()` is chained to `print_sum()`: `print_sum()` coroutine waits until `compute()` is completed before returning its result.

Sequence diagram of the example:



The “Task” is created by the `AbstractEventLoop.run_until_complete()` method when it gets a coroutine object instead of a task.

The diagram shows the control flow, it does not describe exactly how things work internally. For example, the sleep coroutine creates an internal future which uses `AbstractEventLoop.call_later()` to wake up the task in 1 second.

18.5.3.2. InvalidStateError

exception `asyncio.`**`InvalidStateError`**

The operation is not allowed in this state.

18.5.3.3. TimeoutError

exception `asyncio.`**`TimeoutError`**

The operation exceeded the given deadline.

> **Note:** This exception is different from the builtin `TimeoutError` exception!

18.5.3.4. Future

class `asyncio.`**`Future`**(*, *loop=None*)

This class is *almost* compatible with `concurrent.futures.Future`.

Differences:

- `result()` and `exception()` do not take a timeout argument and raise an exception when the future isn't done yet.
- Callbacks registered with `add_done_callback()` are always called via the event loop's `call_soon()`.
- This class is not compatible with the `wait()` and `as_completed()` functions in the `concurrent.futures` package.

This class is not thread safe.

`cancel`()

Cancel the future and schedule callbacks.

If the future is already done or cancelled, return `False`. Otherwise, change the future's state to cancelled, schedule the callbacks and return `True`.

`cancelled`()

Return `True` if the future was cancelled.

`done`()

Return `True` if the future is done.

Done means either that a result / exception are available, or that the future was cancelled.

`result`()

Return the result this future represents.

If the future has been cancelled, raises `CancelledError`. If the future's result isn't yet available, raises `InvalidStateError`. If the future is done and has an exception set, this exception is raised.

`exception`()

Return the exception that was set on this future.

The exception (or `None` if no exception was set) is returned only if the future is done. If the future has been cancelled, raises `CancelledError`. If the future isn't done yet, raises `InvalidStateError`.

`add_done_callback`(*fn*)

Add a callback to be run when the future becomes done.

The callback is called with a single argument - the future object. If the future is already done when this is called, the callback is scheduled with `call_soon()`.

Use functools.partial to pass parameters to the callback. For example, `fut.add_done_callback(functools.partial(print, "Future:", flush=True))` will call `print("Future:", fut, flush=True)`.

remove_done_callback(*fn*)

Remove all instances of a callback from the "call when done" list.

Returns the number of callbacks removed.

set_result(*result*)

Mark the future done and set its result.

If the future is already done when this method is called, raises `InvalidStateError`.

set_exception(*exception*)

Mark the future done and set an exception.

If the future is already done when this method is called, raises `InvalidStateError`.

18.5.3.4.1. Example: Future with run_until_complete()

Example combining a `Future` and a coroutine function:

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

The coroutine function is responsible for the computation (which takes 1 second) and it stores the result into the future. The `run_until_complete()` method waits for the completion of the future.

Note: The `run_until_complete()` method uses internally the `add_done_callback()` method to be notified when the future is done.

18.5.3.4.2. Example: Future with run_forever()

The previous example can be written differently using the `Future.add_done_callback()` method to describe explicitly the control flow:

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

In this example, the future is used to link `slow_operation()` to `got_result()`: when `slow_operation()` is done, `got_result()` is called with the result.

18.5.3.5. Task

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

Schedule the execution of a coroutine: wrap it in a future. A task is a subclass of `Future`.

A task is responsible for executing a coroutine object in an event loop. If the wrapped coroutine yields from a future, the task suspends the execution of the wrapped coroutine and waits for the completion of the future. When the future is done, the execution of the wrapped coroutine restarts with the result or the exception of the future.

Event loops use cooperative scheduling: an event loop only runs one task at a time. Other tasks may run in parallel if other event loops are running in different threads. While a task waits for the completion of a future, the event loop executes a new task.

The cancellation of a task is different from the cancelation of a future. Calling `cancel()` will throw a `CancelledError` to the wrapped coroutine. `cancelled()`

only returns `True` if the wrapped coroutine did not catch the `CancelledError` exception, or raised a `CancelledError` exception.

If a pending task is destroyed, the execution of its wrapped coroutine did not complete. It is probably a bug and a warning is logged: see Pending task destroyed.

Don't directly create `Task` instances: use the `ensure_future()` function or the `AbstractEventLoop.create_task()` method.

This class is not thread safe.

classmethod **all_tasks**(*loop=None*)

Return a set of all tasks for an event loop.

By default all tasks for the current event loop are returned.

classmethod **current_task**(*loop=None*)

Return the currently running task in an event loop or `None`.

By default the current task for the current event loop is returned.

`None` is returned when called not in the context of a `Task`.

cancel()

Request that this task cancel itself.

This arranges for a `CancelledError` to be thrown into the wrapped coroutine on the next cycle through the event loop. The coroutine then has a chance to clean up or even deny the request using try/except/finally.

Unlike `Future.cancel()`, this does not guarantee that the task will be cancelled: the exception might be caught and acted upon, delaying cancellation of the task or preventing cancellation completely. The task may also return a value or raise a different exception.

Immediately after this method is called, `cancelled()` will not return `True` (unless the task was already cancelled). A task will be marked as cancelled when the wrapped coroutine terminates with a `CancelledError` exception (even if `cancel()` was not called).

get_stack(*, *limit=None*)

Return the list of stack frames for this task's coroutine.

If the 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.

The optional limit gives the maximum number of frames to return; by default all available frames are returned. Its meaning 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.)

For reasons beyond our control, only one stack frame is returned for a suspended coroutine.

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

Print the stack or traceback for this task's coroutine.

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

18.5.3.5.1. Example: Parallel execution of tasks

Example executing 3 tasks (A, B, C) in parallel:

```
import asyncio

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

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(
    factorial("A", 2),
    factorial("B", 3),
    factorial("C", 4),
))
loop.close()
```

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
```

A task is automatically scheduled for execution when it is created. The event loop stops when all tasks are done.

18.5.3.6. Task functions

Note: In the functions below, the optional *loop* argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is used.

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

Return an iterator whose values, when waited for, are `Future` instances.

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

Example:

```
for f in as_completed(fs):
    result = yield from f  # The 'yield from' may raise
    # Use result
```

Note: The futures `f` are not necessarily members of fs.

`asyncio.`**`ensure_future`**(*coro_or_future*, *, *loop=None*)

Schedule the execution of a coroutine object: wrap it in a future. Return a `Task` object.

If the argument is a `Future`, it is returned directly.

New in version 3.4.4.

Changed in version 3.5.1: The function accepts any awaitable object.

See also: The `AbstractEventLoop.create_task()` method.

asyncio.**async**(*coro_or_future*, *, *loop=None*)

A deprecated alias to `ensure_future()`.

Deprecated since version 3.4.4.

asyncio.**wrap_future**(*future*, *, *loop=None*)

Wrap a `concurrent.futures.Future` object in a `Future` object.

asyncio.**gather**(**coros_or_futures*, *loop=None*, *return_exceptions=False*)

Return a future aggregating results from the given coroutine objects or futures.

All futures must share the same event loop. If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival). If *return_exceptions* is true, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

Cancellation: if the outer Future is cancelled, all children (that have not completed yet) are also cancelled. If any child is cancelled, this is treated as if it raised `CancelledError` – the outer Future is *not* cancelled in this case. (This is to prevent the cancellation of one child to cause other children to be cancelled.)

Changed in version 3.6.6: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

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

Return `True` if *obj* is a coroutine object, which may be based on a generator or an `async def` coroutine.

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

Return `True` if *func* is determined to be a coroutine function, which may be a decorated generator function or an `async def` function.

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

Submit a coroutine object to a given event loop.

Return a `concurrent.futures.Future` to access the result.

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

```
# 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('The coroutine raised an exception: {!r}'.format(exc))
else:
    print('The coroutine returned: {!r}'.format(result))
```

See the concurrency and multithreading section of the documentation.

Note: Unlike other functions from the module, `run_coroutine_threadsafe()` requires the *loop* argument to be passed explicitly.

New in version 3.5.1.

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

Create a coroutine that completes after a given time (in seconds). If *result* is provided, it is produced to the caller when the coroutine completes.

The resolution of the sleep depends on the granularity of the event loop.

This function is a coroutine.

coroutine `asyncio.`**`shield`**(*arg*, *, *loop=None*)

Wait for a future, shielding it from cancellation.

The statement:

```
res = yield from shield(something())
```

is exactly equivalent to the statement:

```
res = yield from 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. But its caller is still cancelled, so the yield-from expression still raises `CancelledError`. Note: If `something()` is cancelled by other means this will still cancel `shield()`.

If you want to completely ignore cancellation (not recommended) you can combine `shield()` with a try/except clause, as follows:

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

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

Wait for the Futures and coroutine objects given by the sequence *futures* to complete. Coroutines will be wrapped in Tasks. Returns two sets of `Future`: (done, pending).

The sequence *futures* must not be empty.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

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

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.

This function is a coroutine.

Usage:

```
done, pending = yield from asyncio.wait(fs)
```

Note: This does not raise `asyncio.TimeoutError`! Futures that aren't done when the timeout occurs are returned in the second set.

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

Wait for the single `Future` or coroutine object to complete with timeout. If *timeout* is `None`, block until the future completes.

Coroutine will be wrapped in `Task`.

Returns result of the Future or coroutine. When a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`. To avoid the task cancellation, wrap it in `shield()`.

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

This function is a coroutine, usage:

```
result = yield from asyncio.wait_for(fut, 60.0)
```

Changed in version 3.4.3: If the wait is cancelled, the future *fut* is now also cancelled.