Coroutines and Tasks

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 1)
>
> Unknown directive type "currentmodule".
>
> ```
> .. currentmodule:: asyncio
> ```

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

Coroutines

:term:`Coroutines <coroutine>` 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":

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 21); *backlink*
>
> Unknown interpreted text role "term".

```
>>> 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 :func:`asyncio.run` function to run the top-level entry point "main()" function (see the above example.)

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 45); *backlink*
  >
  > Unknown interpreted text role "func".

- 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 :func:`asyncio.create_task` function to run coroutines concurrently as asyncio :class:`Tasks <Task>`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 76); *backlink*

  Unknown interpreted text role "func".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 76); *backlink*

  Unknown interpreted text role "class".

  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 :keyword:`await` expression. Many asyncio APIs are designed to accept awaitables.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 112); *backlink*

Unknown interpreted text role "keyword".

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 :keyword:`async def` function;

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 146); *backlink*

  Unknown interpreted text role "keyword".

- a *coroutine object*: an object returned by calling a *coroutine function*.

Tasks

Tasks are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like :func:`asyncio.create_task` the coroutine is automatically scheduled to run soon:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 156); *backlink*

Unknown interpreted text role "func".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 179); *backlink*

Unknown interpreted text role "class".

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:

```
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 :meth:`loop.run_in_executor`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 203); *backlink*

Unknown interpreted text role "meth".

Creating Tasks

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 210)

Unknown directive type "function".

```
.. function:: create_task(coro, *, name=None, context=None)

   Wrap the *coro* :ref:`coroutine <coroutine>` into a :class:`Task`
   and schedule its execution.  Return the Task object.

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

   An optional keyword-only *context* argument allows specifying a
   custom :class:`contextvars.Context` for the *coro* to run in.
   The current context copy is created when no *context* is provided.

   The task is executed in the loop returned by :func:`get_running_loop`,
   :exc:`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 :func:`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())
       ...

   .. important::

      Save a reference to the result of this function, to avoid
      a task disappearing mid execution.

   .. versionadded:: 3.7

   .. versionchanged:: 3.8
      Added the *name* parameter.

   .. versionchanged:: 3.11
      Added the *context* parameter.
```

Sleeping

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 258)

Unknown directive type "coroutinefunction".

```
.. coroutinefunction:: sleep(delay, result=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.

   Setting the delay to 0 provides an optimized path to allow other
   tasks to run. This can be used by long-running functions to avoid
   blocking the event loop for the full duration of the function call.

   .. _asyncio_example_sleep:

   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())


   .. versionchanged:: 3.10
      Removed the *loop* parameter.
```

Running Tasks Concurrently

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 299)

Unknown directive type "awaitablefunction".

```
.. awaitablefunction:: gather(*aws, return_exceptions=False)

   Run :ref:`awaitable objects <asyncio-awaitables>` 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 :exc:`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.

   .. versionchanged:: 3.10
      Removed the *loop* parameter.

   .. _asyncio_example_gather:

   Example::
```

```
        import asyncio

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

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

        asyncio.run(main())

        # Expected output:
        #
        #     Task A: Compute factorial(2), currently i=2...
        #     Task B: Compute factorial(3), currently i=2...
        #     Task C: Compute factorial(4), currently i=2...
        #     Task A: factorial(2) = 2
        #     Task B: Compute factorial(3), currently i=3...
        #     Task C: Compute factorial(4), currently i=3...
        #     Task B: factorial(3) = 6
        #     Task C: Compute factorial(4), currently i=4...
        #     Task C: factorial(4) = 24
        #     [2, 6, 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.

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

    .. versionchanged:: 3.10
       Removed the *loop* parameter.

    .. deprecated:: 3.10
       Deprecation warning is emitted if no positional arguments are provided
       or not all positional arguments are Future-like objects
       and there is no running event loop.
```

Shielding From Cancellation

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 394)

Unknown directive type "awaitablefunction".

```
.. awaitablefunction:: shield(aw)

   Protect an :ref:`awaitable object <asyncio-awaitables>`
   from being :meth:`cancelled <Task.cancel>`.

   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 :exc:`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

.. versionchanged:: 3.10
   Removed the *loop* parameter.

.. deprecated:: 3.10
   Deprecation warning is emitted if *aw* is not Future-like object
   and there is no running event loop.
```

Timeouts

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 438)

Unknown directive type "coroutinefunction".

```
.. coroutinefunction:: wait_for(aw, timeout)

   Wait for the *aw* :ref:`awaitable <asyncio-awaitables>`
   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
   :exc:`TimeoutError`.

   To avoid the task :meth:`cancellation <Task.cancel>`,
   wrap it in :func:`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.

   .. versionchanged:: 3.10
      Removed the *loop* parameter.

   .. _asyncio_example_waitfor:

   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 TimeoutError:
               print('timeout!')

       asyncio.run(main())
```

```
        # Expected output:
        #
        #     timeout!

    .. versionchanged:: 3.7
       When *aw* is cancelled due to a timeout, ``wait_for`` waits
       for *aw* to be cancelled.  Previously, it raised
       :exc:`TimeoutError` immediately.

    .. versionchanged:: 3.10
       Removed the *loop* parameter.
```

Waiting Primitives

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 498)

Unknown directive type "coroutinefunction".

```
.. coroutinefunction:: wait(aws, *, timeout=None, return_when=ALL_COMPLETED)

   Run :class:`~asyncio.Future` and :class:`~asyncio.Task` instances 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 :exc:`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:

   .. tabularcolumns:: |l|L|

   +-----------------------------+----------------------------------------+
   | Constant                    | Description                            |
   +=============================+========================================+
   | :const:`FIRST_COMPLETED`    | The function will return when any      |
   |                             | future finishes or is cancelled.       |
   +-----------------------------+----------------------------------------+
   | :const:`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     |
   |                             | :const:`ALL_COMPLETED`.                |
   +-----------------------------+----------------------------------------+
   | :const:`ALL_COMPLETED`      | The function will return when all      |
   |                             | futures finish or are cancelled.       |
   +-----------------------------+----------------------------------------+

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

   .. versionchanged:: 3.10
      Removed the *loop* parameter.

   .. versionchanged:: 3.11
      Passing coroutine objects to ``wait()`` directly is forbidden.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 549)

Unknown directive type "function".

```
.. function:: as_completed(aws, *, timeout=None)

   Run :ref:`awaitable objects <asyncio-awaitables>` 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 :exc:`TimeoutError` if the timeout occurs before
   all Futures are done.

   .. versionchanged:: 3.10
      Removed the *loop* parameter.

   Example::

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

   .. versionchanged:: 3.10
      Removed the *loop* parameter.

   .. deprecated:: 3.10
      Deprecation warning is emitted if not all awaitable objects in the *aws*
      iterable are Future-like objects and there is no running event loop.
```

Running in Threads

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 579)

Unknown directive type "coroutinefunction".

```
.. coroutinefunction:: 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 :class:`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 run 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 :term:`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.

   .. versionadded:: 3.9
```

Scheduling From Other Threads

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 638)

Unknown directive type "function".

```
.. function:: run_coroutine_threadsafe(coro, loop)

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

   Return a :class:`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 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 :ref:`concurrency and multithreading <asyncio-multithreading>`
   section of the documentation.

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

   .. versionadded:: 3.5.1
```

Introspection

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 684)

Unknown directive type "function".

```
.. function:: current_task(loop=None)

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

   If *loop* is ``None`` :func:`get_running_loop` is used to get
   the current loop.
```

```
   .. versionadded:: 3.7
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 695)

Unknown directive type "function".

```
.. function:: all_tasks(loop=None)

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

   If *loop* is ``None``, :func:`get_running_loop` is used for getting
   current loop.

   .. versionadded:: 3.7
```

Task Object

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]asyncio-task.rst, line 709)

Invalid class attribute value for "class" directive: "Task(coro, *, loop=None, name=None)".

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

   A :class:`Future-like <Future>` object that runs a Python
   :ref:`coroutine <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 :func:`asyncio.create_task` function to create
   Tasks, or the low-level :meth:`loop.create_task` or
   :func:`ensure_future` functions.  Manual instantiation of Tasks
   is discouraged.

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

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

   :class:`asyncio.Task` inherits from :class:`Future` all of its
   APIs except :meth:`Future.set_result` and
   :meth:`Future.set_exception`.

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

   .. versionchanged:: 3.7
      Added support for the :mod:`contextvars` module.

   .. versionchanged:: 3.8
      Added the *name* parameter.

   .. deprecated:: 3.10
      Deprecation warning is emitted if *loop* is not specified
      and there is no running event loop.
```

```
.. method:: cancel(msg=None)

   Request the Task to be cancelled.

   This arranges for a :exc:`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 :keyword:`try` ...
   ... ``except CancelledError`` ... :keyword:`finally` block.
   Therefore, unlike :meth:`Future.cancel`, :meth:`Task.cancel` does
   not guarantee that the Task will be cancelled, although
   suppressing cancellation completely is not common and is actively
   discouraged.

   .. versionchanged:: 3.9
      Added the *msg* parameter.

   .. deprecated-removed:: 3.11 3.14
      *msg* parameter is ambiguous when multiple :meth:`cancel`
      are called with different cancellation messages.
      The argument will be removed.

   .. _asyncio_example_task_cancel:

   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

.. method:: cancelled()

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

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

.. method:: 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.

.. method:: 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 :exc:`CancelledError` exception.

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

   .. method:: 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
      :exc:`CancelledError` exception.

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

   .. method:: 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 :meth:`Future.add_done_callback`
      for more details.

   .. method:: 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 :meth:`Future.remove_done_callback`
      for more details.

   .. method:: 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.)

   .. method:: 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 :meth:`get_stack`.

      The *limit* argument is passed to :meth:`get_stack` directly.

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

   .. method:: get_coro()

      Return the coroutine object wrapped by the :class:`Task`.

      .. versionadded:: 3.8
```

```
      .. method:: 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.

         .. versionadded:: 3.8

      .. method:: 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 :func:`repr` output of a task object.

         .. versionadded:: 3.8
```