


- [index](#)
- [modules](#) |
- [suivant](#) |
- [précédent](#) |
- 

- [Python](#) »

- French

▼

3.9.7

▼

[Documentation](#) »

- [La bibliothèque standard](#) »

- [Réseau et communication entre processus](#) »

- [asyncio — Entrées/Sorties asynchrones](#) »

- 

|

## Coroutines et tâches

Cette section donne un aperçu des API de haut-niveau du module *asyncio* pour utiliser les coroutines et les tâches.

- [Coroutines](#)
- [Attendables](#)
- [Exécution d'un programme \*asyncio\*](#)
- [Création de tâches](#)
- [Attente](#)
- [Exécution de tâches de manière concurrente](#)
- [Protection contre l'annulation](#)
- [Délais d'attente](#)
- [Primitives d'attente](#)
- [Running in Threads](#)
- [Planification depuis d'autres fils d'exécution](#)
- [Introspection](#)
- [Objets \*Task\*](#)
- [Coroutines basées sur des générateurs](#)

## Coroutines

Les [coroutines](#) déclarées avec la syntaxe *async/await* sont la manière privilégiée d'écrire des applications *asyncio*. Par exemple, l'extrait de code suivant (requiert Python 3.7+) affiche « hello », attend 1 seconde et affiche ensuite « world » :

```
>>>
>>> import asyncio

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

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

Appeler une coroutine ne la planifie pas pour exécution :

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

Pour réellement exécuter une coroutine, *asyncio* fournit trois mécanismes principaux :

- La fonction `asyncio.run()` pour exécuter la fonction « `main()` », le point d'entrée de haut-niveau (voir l'exemple ci-dessus).
- Attendre une coroutine. Le morceau de code suivant attend une seconde, affiche « `hello` », attend 2 secondes *supplémentaires*, puis affiche enfin « `world` » :

```

• 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())

```

Sortie attendue :

```

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

```

- La fonction `asyncio.create_task()` pour exécuter de manière concurrente des coroutines en tant que *tâches* `asyncio`.

Modifions l'exemple ci-dessus et lançons deux coroutines `say_after` de manière concurrente :

```

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')}")

```

La sortie attendue montre à présent que ce code s'exécute une seconde plus rapidement que le précédent :

```

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

```

## Attendables

Un objet est dit *attendable* (*awaitable* en anglais, c.-à-d. qui peut être attendu) s'il peut être utilisé dans une expression `await`. Beaucoup d'API d'*asyncio* sont conçues pour accepter des *attendables*.

Il existe trois types principaux d'*attendables* : les **coroutines**, les **tâches** et les **futurs**.

### Coroutines

Les coroutines sont des *awaitables* et peuvent donc être attendues par d'autres 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

Dans cette documentation, le terme « coroutine » est utilisé pour désigner deux concepts voisins :

- une *fonction coroutine* : une fonction `async def` ;
- un *objet coroutine* : un objet renvoyé par une *fonction coroutine*.

`asyncio` implémente également les coroutines [basées sur des générateurs](#) ; celles-ci sont obsolètes.

## Tâches

Les *tâches* servent à planifier des coroutines de façon à ce qu'elles s'exécutent de manière concurrente.

Lorsqu'une coroutine est encapsulée dans une *tâche* à l'aide de fonctions comme `asyncio.create_task()`, la coroutine est automatiquement planifiée pour s'exécuter prochainement :

```
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())
```

## Futurs

Un `Future` est un objet *awaitable* spécial de **bas-niveau**, qui représente le **résultat final** d'une opération asynchrone.

Quand un objet *Future* est *attendu*, cela signifie que la coroutine attendra que ce futur soit résolu à un autre endroit.

Les objets *Future* d'*asyncio* sont nécessaires pour permettre l'exécution de code basé sur les fonctions de rappel avec la syntaxe *async* / *await*.

Il est normalement **inutile** de créer des objets *Future* dans la couche applicative du code.

Les objets *Future*, parfois exposés par des bibliothèques et quelques API d'*asyncio*, peuvent être attendus :

```
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()
    )
```

`loop.run_in_executor()` est l'exemple typique d'une fonction bas-niveau renvoyant un objet *Future*.

## Exécution d'un programme *asyncio*

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

Exécute la `coroutine` `coro` et renvoie le résultat.

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

Cette fonction ne peut pas être appelée si une autre boucle d'événement `asyncio` s'exécute dans le même fil d'exécution.

Si `debug` vaut `True`, la boucle d'événement s'exécute en mode de débogage.

Cette fonction crée toujours une nouvelle boucle d'événement et la clôt à la fin. Elle doit être utilisée comme point d'entrée principal des programmes `asyncio` et ne doit être idéalement appelée qu'une seule fois.

Exemple :

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

asyncio.run(main())
```

*Nouveau dans la version 3.7.*

*Modifié dans la version 3.9:* Updated to use `loop.shutdown_default_executor()`.

#### Note

Le code source pour `asyncio.run()` est disponible dans [Lib/asyncio/runners.py](https://lib/asyncio/runners.py).

## Création de tâches

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

Encapsule la `coroutine` `coro` dans une tâche et la planifie pour exécution. Renvoie l'objet `Task`.

Si `name` n'est pas `None`, il est défini comme le nom de la tâche en utilisant `Task.set_name()`.

La tâche est exécutée dans la boucle renvoyée

par `get_running_loop()` ; `RuntimeError` est levée s'il n'y a pas de boucle en cours d'exécution dans le fil actuel.

Cette fonction a été **ajoutée dans Python 3.7**. Pour les versions antérieures à la 3.7, la fonction `asyncio.ensure_future()` de bas-niveau `asyncio.ensure_future()` peut-être utilisée :

```
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())
...
```

*Nouveau dans la version 3.7.*

*Modifié dans la version 3.8:* ajout du paramètre `name`.

## Attente

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

Attend pendant *delay* secondes.

Si *result* est spécifié, il est renvoyé à l'appelant quand la coroutine se termine.

`sleep()` suspend systématiquement la tâche courante, ce qui permet aux autres tâches de s'exécuter.



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.

*Deprecated since version 3.8, will be removed in version 3.10: Le paramètre loop.*

Exemple d'une coroutine affichant la date toutes les secondes pendant 5 secondes :

```
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())
```

## Exécution de tâches de manière concurrente

*awaitable* asyncio.gather(\*aws, loop=None, return\_exceptions=False)

Exécute les objets *awaitable* de la séquence *aws*, de manière concurrente.

Si un *attendable* de *aws* est une coroutine, celui-ci est automatiquement planifié comme une tâche *Task*.

Si tous les *awaitables* s'achèvent avec succès, le résultat est la liste des valeurs

renvoyées. L'ordre de cette liste correspond à l'ordre des *awaitables* dans *aws*.

Si *return\_exceptions* vaut `False` (valeur par défaut), la première exception levée est immédiatement propagée vers la tâche en attente dans le `gather()`. Les autres *awaitables* dans la séquence *aws* **ne sont pas annulés** et poursuivent leur exécution.

Si *return\_exceptions* vaut `True`, les exceptions sont traitées de la même manière que les exécutions normales, et incluses dans la liste des résultats.

Si `gather()` est *annulé*, tous les *awaitables* en cours (ceux qui n'ont pas encore fini de s'exécuter) sont également *annulés*.

Si n'importe quel *Task* ou *Future* de la séquence *aws* est *annulé*, il est traité comme s'il avait levé `CancelledError` — l'appel à `gather()` n'est alors **pas** annulé. Ceci permet d'empêcher que l'annulation d'une tâche ou d'un futur entraîne l'annulation des autres tâches ou futurs.

*Deprecated since version 3.8, will be removed in version 3.10:* Le paramètre *loop*.

Exemple :

```
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.

*Modifié dans la version 3.7:* Si *gather* est lui-même annulé, l'annulation est propagée indépendamment de *return\_exceptions*.

## Protection contre l'annulation

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

Empêche qu'un objet `awaitable` puisse être annulé.

Si *aw* est une coroutine, elle est planifiée automatiquement comme une tâche.

L'instruction :

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

est équivalente à :

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

à la différence près que, si la coroutine qui la contient est annulée, la tâche s'exécutant dans `something()` n'est pas annulée. Du point de vue de `something()`, il n'y a pas eu d'annulation. Cependant, son appelant est bien annulé, donc l'expression *await* lève bien une `CancelledError`.

Si `something()` est annulée d'une autre façon (c.-à-d. depuis elle-même) ceci annule également `shield()`.

Pour ignorer complètement l'annulation (déconseillé), la fonction `shield()` peut être combinée à une clause *try / except*, comme dans le code ci-dessous :

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

*Deprecated since version 3.8, will be removed in version 3.10: Le paramètre loop.*

## Délais d'attente

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

Attend la fin de l'[awaitable](#) `aw` avec délai d'attente.

Si `aw` est une coroutine, elle est planifiée automatiquement comme une tâche.

`timeout` peut-être soit `None`, soit le nombre de secondes (entier ou décimal) d'attente. Si `timeout` vaut `None`, la fonction s'interrompt jusqu'à ce que le futur s'achève.

Si le délai d'attente maximal est dépassé, la tâche est annulée et l'exception `asyncio.TimeoutError` est levée.

Pour empêcher l'[annulation](#) de la tâche, il est nécessaire de l'encapsuler dans une fonction `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.

Si l'attente est annulée, le futur `aw` est également annulé.

*Deprecated since version 3.8, will be removed in version 3.10: Le paramètre loop.*

Exemple :

```
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!
```

Modifié dans la version 3.7: Si le dépassement du délai d'attente maximal provoque l'annulation de `aw`, `wait_for` attend que `aw` soit annulée. Auparavant, l'exception `asyncio.TimeoutError` était immédiatement levée.

## Primitives d'attente

*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.

Renvoie deux ensembles de *Tasks / Futures*: (done, pending).

Utilisation :

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

*timeout* (entier ou décimal), si précisé, peut-être utilisé pour contrôler le nombre maximal de secondes d'attente avant de se terminer.

Cette fonction ne lève pas `asyncio.TimeoutError`. Les futurs et les tâches qui ne sont pas finis quand le délai d'attente maximal est dépassé sont tout simplement renvoyés dans le second ensemble.

*return\_when* indique quand la fonction doit se terminer. Il peut prendre les valeurs suivantes :

Constante	Description
FIRST_COMPLETED	La fonction se termine lorsque n'importe quel futur se termine ou est annulé.
FIRST_EXCEPTION	La fonction se termine lorsque n'importe quel futur se termine en levant une exception. Si aucun <i>futur</i> ne lève d'exception, équivaut à ALL_COMPLETED.
ALL_COMPLETED	La fonction se termine lorsque les <i>futurs</i> sont tous finis ou annulés.

À la différence de `wait_for()`, `wait()` n'annule pas les futurs quand le délai d'attente est dépassé.

*Obsolète depuis la version 3.8:* Si un *awaitable* de *aws* est une coroutine, celle-ci est automatiquement planifiée comme une tâche. Passer directement des objets coroutines à `wait()` est obsolète, car ceci conduisait à un comportement portant à confusion.

*Deprecated since version 3.8, will be removed in version 3.10: Le paramètre loop.*

#### Note

`wait()` planifie automatiquement les coroutines comme des tâches et renvoie les objets `Task` ainsi créés dans les ensembles `(done, pending)`. Le code suivant ne fonctionne donc pas comme voulu :

```
async def foo():
    return 42

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

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

Voici comment corriger le morceau de code ci-dessus :

```
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: Passer directement des objets coroutines à `wait()` est obsolète.*

```
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.

Lève une exception `asyncio.TimeoutError` si le délai d'attente est dépassé avant que tous les futurs ne soient achevés.

*Deprecated since version 3.8, will be removed in version 3.10: Le paramètre `loop`.*

Exemple :

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

## Running in Threads

```
coroutine asyncio
o.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.

*Nouveau dans la version 3.9.*

## Planification depuis d'autres fils d'exécution

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

Enregistre une coroutine dans la boucle d'exécution actuelle. Cette opération est compatible avec les programmes à multiples fils d'exécution (*thread-safe*).

Renvoie

un `concurrent.futures.Future` pour attendre le résultat d'un autre fil d'exécution du système d'exploitation.

Cette fonction est faite pour être appelée par un fil d'exécution distinct de celui dans laquelle la boucle d'événement s'exécute. Exemple :

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

```

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

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

```

Si une exception est levée dans une coroutine, le futur renvoyé en sera averti. Elle peut également être utilisée pour annuler la tâche de la boucle d'événement :

```

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}')

```

Voir la section [exécution concurrente et multi-fils d'exécution](#) de la documentation.

À la différence des autres fonctions d'*asyncio*, cette fonction requiert que *loop* soit passé de manière explicite.

*Nouveau dans la version 3.5.1.*

[Intros](#)  
[pectio](#)  
[n](#)  
 asyncio  
**.curre**

```
nt_tasks  
k(loop=  
None)
```

Renvoie l'instance de la `Task` en cours d'exécution, ou `None` s'il n'y a pas de tâche en cours.

Si *loop* vaut `None`, `get_running_loop()` est appelée pour récupérer la boucle en cours d'exécution.

*Nouveau dans la version 3.7.*

```
asyncio.  
all_tasks(  
loop=  
None)
```

Renvoie l'ensemble des `Task` non terminés en cours d'exécution dans la boucle.

Si *loop* vaut `None`, `get_running_loop()` est appelée pour récupérer la boucle en cours d'exécution.

*Nouveau dans la version 3.7.*

O  
b  
j  
e  
t  
s  
  
T  
a  
s

*k*

*c/*  
*a*  
*s*  
*s*

*a*  
*s*  
*y*  
*n*  
*c*  
*i*  
*o*

**. Task**  
**(**  
*c*  
*o*  
*r*  
*o*  
**,**  
*\**  
**,**  
*l*  
*o*  
*o*  
*p*  
**=**  
*N*  
*o*  
*n*  
*e*  
**,**  
*n*  
*a*  
*m*  
*e*  
**=**  
*N*  
*o*  
*n*  
*e*  
**)**

Objet compatible avec `Future` qui exécute une `coroutine` Python. Cet objet n'est pas utilisable dans des programmes à fils d'exécution multiples.

Les tâches servent à exécuter des coroutines dans des boucles d'événements. Si une coroutine attend un futur, la tâche interrompt son exécution et attend la fin de ce *futur*. Quand celui-ci est terminé, l'exécution de la coroutine encapsulée reprend.

Les boucles d'événement fonctionnent de manière *coopérative* : une boucle d'événement exécute une tâche à la fois. Quand une tâche attend la fin d'un futur, la boucle d'événement exécute d'autres tâches, des fonctions de rappel, ou effectue des opérations d'entrées-sorties.

La fonction de haut niveau `asyncio.create_task()` et les fonctions de bas-niveau `loop.create_task()` ou `ensure_future()` permettent de créer des tâches. Il est déconseillé d'instancier manuellement des objets *Task*.

La méthode `cancel()` d'une tâche en cours d'exécution permet d'annuler celle-ci. L'appel de cette méthode force la tâche à lever l'exception `CancelledError` dans la coroutine encapsulée. Si la coroutine attendait un *futur* au moment de l'annulation, celui-ci est annulé.

La méthode `cancelled()` permet de vérifier si la tâche a été annulée. Elle renvoie `True` si la coroutine encapsulée n'a pas ignoré l'exception `CancelledError` et a bien été annulée.

`asyncio.Task` hérite de `Future`, de toute son API, à l'exception de `Future.set_result()` et de `Future.set_exception()`.

`Task` implémente le module `contextvars`. Lors de sa création, une tâche effectue une copie du contexte actuel et exécutera ses coroutines dans cette copie.

*Modifié dans la version 3.7:* Ajout du support du module `contextvars`.

*Modifié dans la version 3.8:* ajout du paramètre `name`.

*Deprecated since version 3.8, will be removed in version 3.10:* Le paramètre `loop`.

**`cancel(msg=None)`**

Demande l'annulation d'une tâche.

Provisionne la levée de l'exception `CancelledError` dans la coroutine encapsulée. L'exception sera levée au prochain cycle de la boucle d'exécution.

La coroutine peut alors faire le ménage ou même ignorer la requête en supprimant l'exception à l'aide d'un bloc `try... except CancelledError ... finally`.

Par conséquent, contrairement à `Future.cancel()`, `Task.cancel()` ne garantit pas que la tâche sera annulée, bien qu'ignorer totalement une annulation ne soit ni une pratique courante, ni encouragée.

*Modifié dans la version 3.9:* Added the `msg` parameter.

L'exemple ci-dessous illustre comment une coroutine peut intercepter une requête d'annulation :



```

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()

Renvoie `True` si la tâche est *annulée*.

La tâche est *annulée* quand l'annulation a été demandée avec `cancel()` et la coroutine encapsulée a propagé

l'exception `CancelledError` qui a été levée en son sein.

### **done()**

Renvoie `True` si la tâche est *achevée*.

Une tâche est dite *achevée* quand la coroutine encapsulée a soit renvoyé une valeur, soit levé une exception, ou que la tâche a été annulée.

### **result()**

Renvoie le résultat de la tâche.

Si la tâche est *achevée*, le résultat de la coroutine encapsulée est renvoyé (sinon, dans le cas où la coroutine a levé une exception, cette exception est de nouveau levée).

Si la tâche a été *annulée*, cette méthode lève une exception `CancelledError`.

Si le résultat de la tâche n'est pas encore disponible, cette méthode lève une exception `InvalidStateError`.

### **exception()**

Renvoie l'exception de la tâche.

Si la coroutine encapsulée lève une exception, cette exception est renvoyée. Si la coroutine s'est exécutée normalement, cette méthode renvoie `None`.

Si la tâche a été *annulée*, cette méthode lève une exception `CancelledError`.

Si la tâche n'est pas encore *achevée*, cette méthode lève une exception `InvalidStateError`.

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

Ajoute une fonction de rappel qui sera exécutée quand la tâche sera *achevée*.

Cette méthode ne doit être utilisée que dans du code basé sur les fonctions de rappel de bas-niveau.

Se référer à la documentation de `Future.add_done_callback()` pour plus de détails.

```
remove_done_callback(callback)
```

Retire *callback* de la liste de fonctions de rappel.

Cette méthode ne doit être utilisée que dans du code basé sur les fonctions de rappel de bas-niveau.

Se référer à la documentation de `Future.remove_done_callback()` pour plus de détails.

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

Renvoie une liste représentant la pile d'appels de la tâche.

Si la coroutine encapsulée n'est pas terminée, cette fonction renvoie la pile d'appels à partir de l'endroit où celle-ci est interrompue. Si la coroutine s'est terminée normalement ou a été annulée, cette fonction renvoie une liste vide. Si la coroutine a été terminée par une exception, ceci renvoie la pile d'erreurs.

La pile est toujours affichée de l'appelant à l'appelé.

Une seule ligne est renvoyée si la coroutine est suspendue.

L'argument facultatif *limit* définit le nombre maximal d'appels à renvoyer ; par défaut, tous sont renvoyés. L'ordre de la liste diffère selon la nature de celle-ci : les appels les plus récents d'une pile d'appels sont renvoyés, si la pile est une pile d'erreurs, ce sont les appels les plus anciens qui le sont (dans un souci de cohérence avec le module *traceback*).

```
print_s  
tack(*, l  
      imit=Non  
      e, file=N  
      one)
```

Affiche la pile d'appels ou d'erreurs de la tâche.

Le format de sortie des appels produits par `get_stack()` est similaire à celui du module *traceback*.

Le paramètre *limit* est directement passé à `get_stack()`.

Le paramètre *file* est un flux d'entrées-sorties sur lequel le résultat est écrit ; par défaut, `sys.stderr`.

```
get_  
coro(  
)
```

Renvoie l'objet *coroutine* encapsulé par la `Task`.

*Nouveau dans la version 3.8.*

**ge  
t\_  
na  
me  
( )**

Renvoie le nom de la tâche.

Si aucun nom n'a été explicitement assigné à la tâche, l'implémentation par défaut d'une *Task asyncio* génère un nom par défaut durant l'instanciation.

*Nouveau dans la version 3.8.*

**s  
e  
t  
\_  
n  
a  
m  
e  
(  
v  
a  
l  
u  
e  
)**

Définit le nom de la tâche.

L'argument *value* peut être n'importe quel objet qui sera ensuite converti en chaîne de caractères.

Dans l'implémentation par défaut de *Task*, le nom sera visible dans le résultat de `repr()` d'un objet *Task*.

*Nouveau dans la version 3.8.*

# Corollaire

r  
s

**N  
o  
t  
e**

L  
e  
s  
c  
o  
r  
o  
u  
t  
i  
n  
e  
s  
b  
a  
s  
é  
e  
s  
s  
u  
r  
d  
e  
s  
g  
é  
n  
é  
r  
a

t  
e  
u  
r  
s  
s  
o  
n  
t

**o  
b  
s  
o  
l  
è  
t  
e  
s**

e  
t  
i  
l  
e  
s  
t  
p  
r  
é  
v  
u  
d  
e  
l  
e  
s  
s  
u  
p  
p



r  
i  
m  
e  
r  
e  
n  
P  
y  
t  
h  
o  
n  
3  
.  
1  
0  
.  
L  
e  
s  
c  
o  
r  
o  
u  
t  
i  
n  
e  
s  
b  
a  
s  
é  
e  
s  
s  
u  
r

d  
e  
s  
g  
é  
n  
é  
r  
a  
t  
e  
u  
r  
s  
s  
o  
n  
t  
a  
n  
t  
é  
r  
i  
e  
u  
r  
e  
s  
à  
l  
a  
s  
y  
n  
t  
a  
x  
e  
  
a

*s  
y  
n  
c*

*/*

*a  
w  
a  
i  
t*

*.  
l  
l  
e  
x  
i  
s  
t  
e  
d  
e  
s  
g  
é  
n  
é  
r  
a  
t  
e  
u  
r  
s*

*P  
y  
t  
h  
o*

*n*

quintililess expressions

yield from

p

o  
u  
r  
a  
t  
t  
e  
n  
d  
r  
e  
d  
e  
s

*f*  
*u*  
*t*  
*u*  
*r*  
s

e  
t  
a  
u  
t  
r  
e  
s  
c  
o  
r  
o  
u  
t  
i  
n  
e  
s  
.

L  
e  
s  
c  
o  
r  
o  
u  
t  
i  
n  
e  
s  
b  
a  
s  
é  
e  
s  
s  
u  
r  
d  
e  
s  
g  
é  
n  
é  
r  
a  
t  
e  
u  
r  
s  
d  
o  
i  
v  
e

n  
t  
ê  
t  
r  
e  
d  
é  
c  
o  
r  
é  
e  
s  
a  
v  
e  
c

@  
a  
s  
y  
n  
c  
i  
o  
.  
c  
o  
r  
o  
u  
t  
i  
n  
e  
,  
m  
ê  
m

e  
s  
i  
c  
e  
n  
,  
e  
s  
t  
p  
a  
s  
v  
é  
r  
i  
f  
i  
é  
p  
a  
r  
l  
,  
i  
n  
t  
e  
r  
p  
r  
é  
t  
e  
u  
r  
.  
@  
a  
s



Décorateur pour coroutines basées sur des générateurs.

Ce décorateur rend compatibles les coroutines basées sur des générateurs avec le code *async / await* :

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

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

Ce décorateur ne doit pas être utilisé avec des coroutines `async def`.

*Deprecated since version 3.8, will be removed in version 3.10: utilisez `async def` à la place.*

Renvoie `True` si *obj* est un [objet coroutine](#).

Cette méthode est différente de `inspect.iscoroutine()` car elle renvoie `True` pour des coroutines basées sur des générateurs.

Renvoie `True` si *func* est une **fonction coroutine**.

Cette méthode est différente de `inspect.iscoroutinefunction()` car elle renvoie `True` pour des coroutines basées sur des générateurs, décorées avec `@coroutine`.