**How to Manage Exceptions When Waiting On Multiple Asyncio Tasks**

**Handling exceptions in asyncio requires careful management to ensure that code does not become unstable.**

This is the second part of a series of articles looking at problems arising when waiting on multiple events in Python Asyncio. The [first article](https://python.plainenglish.io/how-to-avoid-issues-when-waiting-on-multiple-events-in-python-asyncio-48e22d148de7) explored how to manage the lifecycle of tasks when using asyncio.wait. This article discusses how asyncio.wait handles exceptions and what the programmer needs to do to ensure that results are gathered and tasks terminated in a coordinated manner.

**Waiting on a single coroutine**

When awaiting a single coroutine exception handling is managed in exactly the same way as non-asyncio code using a try…except…finally, block. As an example, an exception is raised in the foo coroutine which propagates up to the main block where it is captured and the error is printed.

import asyncio

async def foo():

raise ValueError("Foo value error")

return("Foo finished")

async def main():

try:

result = await foo()

print(result)

except ValueError as e:

print(f"ValueError: {e}")

asyncio.run(main())

ValueError: Foo value error

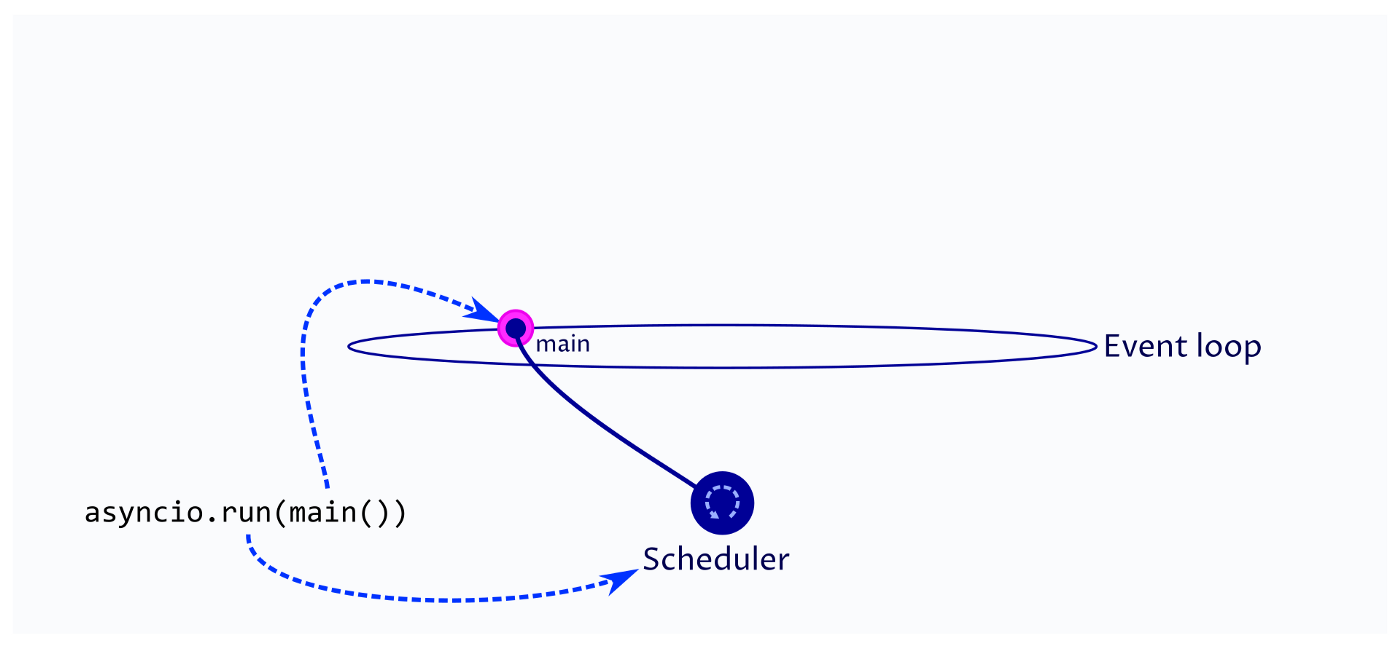
Process finished with exit code 0

However, if the result in main is waiting on several coroutines to complete what should happen if one of those coroutines throws an exception? Should it be immediately propagated to the main coroutine? What about unfinished tasks? Should they be terminated or allowed to continue? To prevent any unwanted side-effects the programmer needs to have a plan of action to handle exceptions when they arise.

**It’s the scheduler that calls the coroutine**

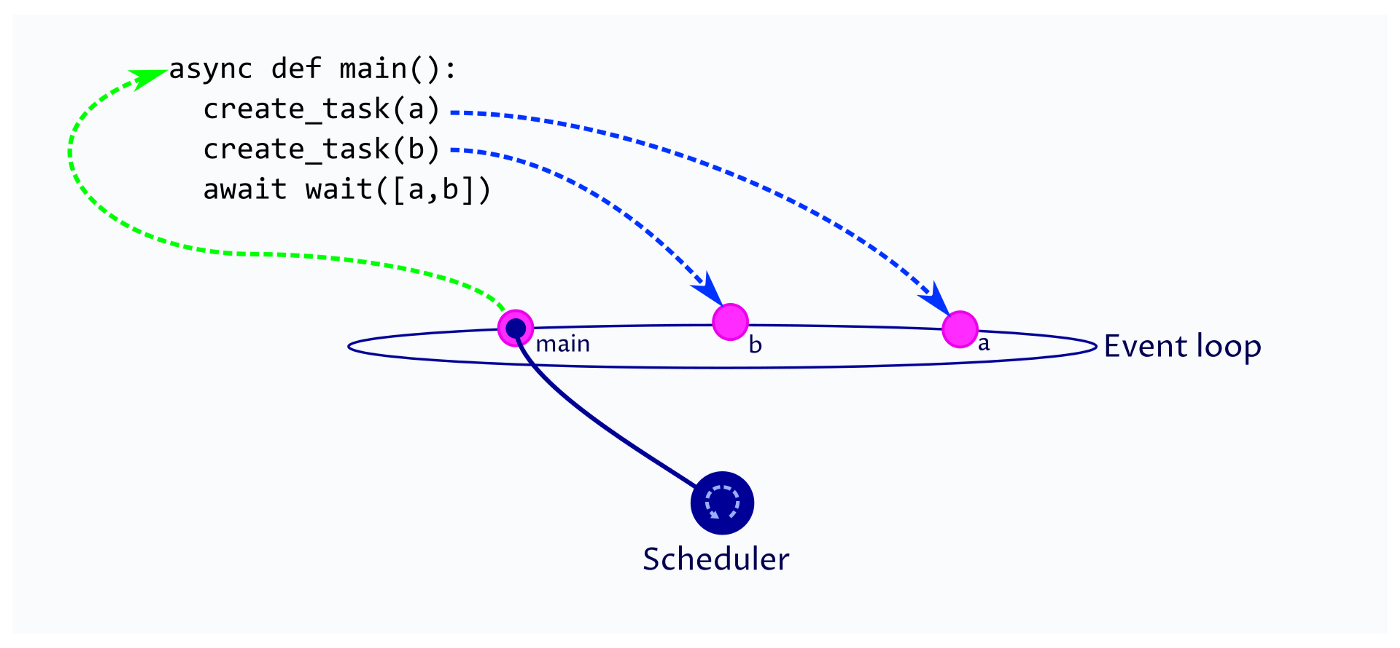
In the previous example when we run the main and await the foo coroutines they don’t start running immediately. Instead, they are added to a list of coroutines to run at some point in time in the future. This is a major difference compared with regular Python functions and what allows multiple tasks to seemingly run in parallel.

So, when asyncio.run(main()) is called the coroutine main() is placed on the event loop and, as it is the only coroutine on the event loop, it is called by the scheduler and starts executing. When the main coroutine meets the await foo() statement the foo coroutine is added to the event list and execution of the main function is suspended with control yielded back to the scheduler. The scheduler then looks to see if there are any other tasks to execute and seeing foo calls the coroutine until it terminates. Termination of foo sets a flag in main that the call to foo has completed; so on the next trip round the event loop, the scheduler sees that main is able to continue execution and so calls it, with execution continuing from the point where main yielded.



asyncio run starts the event loop and creates a task for the scheduler to execute.

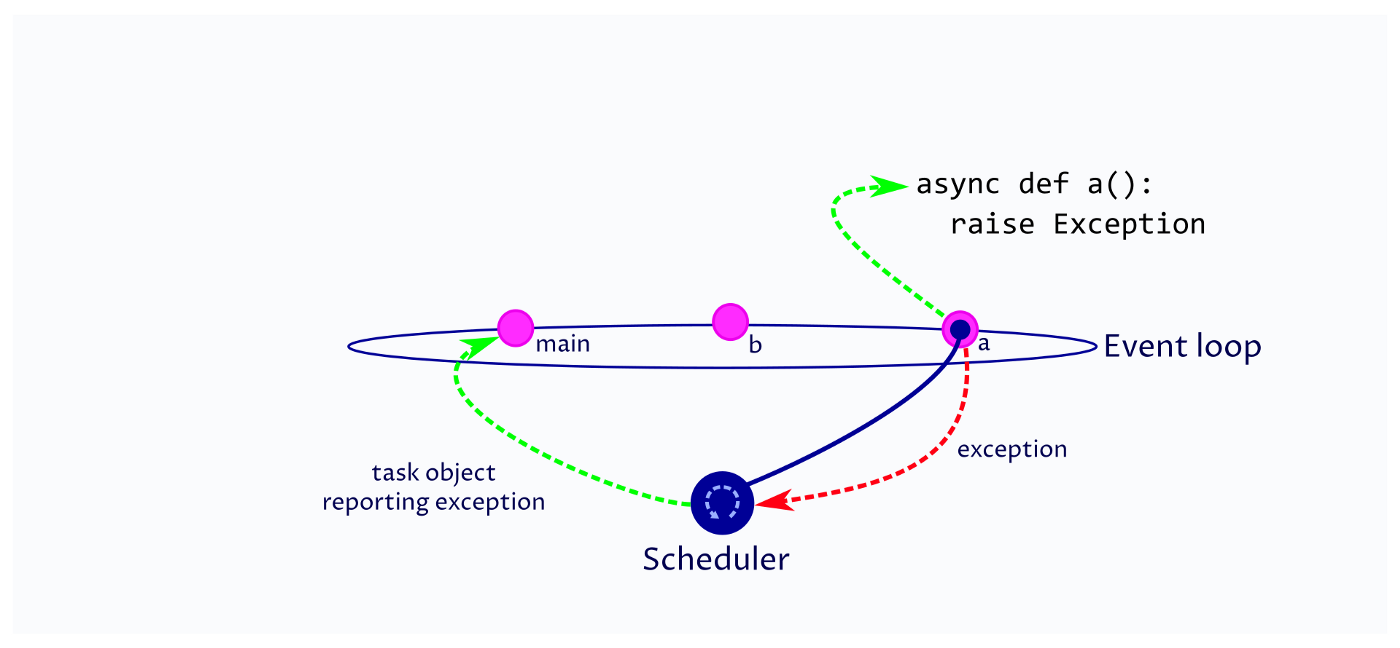
We can also consider the case when main is waiting on two tasks to complete. In this case, two tasks are created (a, b). These are added to the event loop, but will not execute until the main loop yields control back to the scheduler and the scheduler is then free to call the newly created tasks.



The scheduler executes the first task (main) until it hands execution back to the scheduler (await). The task may create additional tasks which are added to the event loop.

We can now consider the case where the main coroutine has yielded control to the scheduler and the scheduler has called task a. For some reason, task a raises an exception. This exception does not propagate up to main, as main is a peer of a on the event loop and does, instead, propagate up to the scheduler.

The scheduler captures the exception and passes a task object back to main noting that an exception has been raised. Depending upon the parameters in asyncio.wait the scheduler will then determine whether main is callable on the next trip around the event loop, or whether all tasks have to complete before main is called.



When a co-routine raises an exception it propagates up to the scheduler. The scheduler captures the exception, marks the task as done, removes it from the event loop and notes the exception in the task object.

At this point you will note:

* No exception has been raised in main at this point. It can only be handled once the scheduler calls the main coroutine — which occurs when the return\_when condition is met.
* The coroutine a has terminated and will be removed from the event loop. All the resources it used will be released.
* The status of coroutine b is not determined. It might or might not have started execution; and, might or might not have completed. If the return\_when condition for main is FIRST\_EXCEPTION then main may continue execution and complete before b is executed and terminates.

When waiting on multiple tasks a plan is needed:

* What behaviour is required when an exception is raised in a task (coroutine)? Do you want to wait for all other tasks to complete before the calling coroutine continues execution or return immediately?
* If you wait for all tasks to return how will you determine which tasks returned results and which had exceptions? How many tasks can simultaneously return exceptions? This could be zero up to the number of tasks that you are waiting on.
* If you return when the first exception is raised how do you ensure that all other tasks have completed execution and any resources they use are safely released? Tasks may be in an unknown state and may create unwanted side-effects if they are left to run indefinitely or resources they rely on are destroyed. For instance, if the calling coroutine closes network connections before the called coroutine is executed then the called coroutine may wait indefinitely for a network connection that doesn’t exist.

**Capturing exceptions raised in tasks**

To demonstrate exception handling with asyncio.wait the following example has two coroutines: foo which immediately raises a ValueError and bar which sleeps for one second and then returns. The main function encapsulates these coroutines in tasks before waiting for both tasks to complete. When the tasks complete then completed tasks are returned in the done list and tasks which have not completed are returned in pending.

import asyncio

async def foo():

raise ValueError("Foo value error")

return("Foo finished")

async def bar():

await asyncio.sleep(1)

return("Bar finished")

async def main():

foo\_task = asyncio.create\_task(foo(), name="Exception task")

bar\_task = asyncio.create\_task(bar(), name="Waiting task")

try:

done, pending = await asyncio.wait(

[foo\_task, bar\_task],

return\_when=asyncio.ALL\_COMPLETED

)

for task in done:

name = task.get\_name()

print(f"Done: {name}")

for task in pending:

task.cancel()

except Exception as e:

print(f"Exception caught: {e}")

asyncio.run(main())

When foo\_task and bar\_task complete the done list contains both tasks. However, at this point the exception raised in foo is not re-thrown into main. If the code is run then the following output is produced.

Done: Waiting task  
Done: Exception task------------ At program termination -------------  
Task exception was never retrieved  
future: <Task   
 finished name='Exception task'  
 coro=<foo() done,exception=ValueError('Foo value error')>  
>ValueError: Foo value error

We can clearly see that both tasks completed, and we are able to retrieve the name of the tasks and print them. However, accessing the name of the task did not cause the exception in foo\_task to be re-thrown. What is perhaps more concerning is that the foo\_task remains on the event loop until the program terminates. If the exception is not handled then tasks can remain on the event loop indefinitely.

**Retrieving exceptions for each task**

Tasks returned from asyncio.wait have the following methods which are relevant to handling exceptions:

* get\_name() : This is optional, however, giving tasks meaningful names makes creating descriptive exceptions and debugging significantly easier.
* exception() : Returns the exception object raised by the coroutine, None if no exception as raised.
* result() : Returns the result of the coroutine, re-throws the exception if the coroutine raised an exception.

Using these methods we can rewrite the main function to print the name of the task; test whether the coroutine threw an exception and print the exception details; and, retrieve the result of the coroutine, using a try…except block to capture any re-thrown exceptions.

import asyncio

async def foo():

raise ValueError("Foo value error")

return("Foo finished")

async def bar():

await asyncio.sleep(1)

return("Bar finished")

async def main():

foo\_task = asyncio.create\_task(foo(), name="Exception\_task")

bar\_task = asyncio.create\_task(bar(), name="Waiting\_task")

try:

done, pending = await asyncio.wait(

[foo\_task, bar\_task],

return\_when=asyncio.ALL\_COMPLETED

)

for task in done:

name = task.get\_name()

print(f"DONE: {name}")

exception = task.exception() \*\*\*\*\*\*\*\* NEW CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*

if isinstance(exception, Exception):

print(f"{name} threw {exception}")

try:

result = task.result()

print(f"{name} returned {result}")

except ValueError as e:

print(f"ValueError: {e}") \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for task in pending:

task.cancel()

except Exception as e:

print("Outer Exception")

asyncio.run(main())

Running the above code produces the following output. We can clearly see that foo threw a ValueError exception, which was then re-thrown and caught when .result() was called. Note, the ordering of the returned tasks is the order in which they completed; and, using this comprehensive approach to handling exceptions and pending tasks ensures that all tasks are managed and removed from the event loop prior to termination of the calling coroutine (in this case main).

DONE: Exception\_task  
Exception\_task threw Foo value error  
ValueError: Foo value error  
DONE: Waiting\_task  
Waiting\_task returned Bar finished------------ At program termination -------------

There are a couple of considerations when planning how to handle exceptions and cancel pending tasks:

* Plan how you will handle pending tasks and exceptions. It is difficult to debug asyncio code when there is no code to explicitly handle these conditions.
* There is a high probability that tasks that throw exceptions will return earlier than those that return a value result. This means they will appear earlier, if not first, in the iterable list. Therefore, the first task for which result() method is called may throw an exception, in which case you may want to process all other results and cancel pending tasks before re-throwing this error.
* The ordering of activities following asyncio.wait may be important in your implementation. Work out the most appropriate order for cancelling pending tasks, getting results, testing for and handling exceptions.
* If more than one task throws an exception you may need to work out how to aggregate them into a single error message that can be re-thrown once all other tasks have been processed.

**Exceptions in asyncio.gather**

Python provides asyncio.gather as a higher-level alternative to asyncio.wait when waiting for all tasks to complete. It manages the wrapping of coroutines in tasks and ensuring the tasks are removed from the event loop when the function returns. Using the previous foo bar example we can replace asyncio.wait with:

results = await asyncio.gather(foo(), bar(), return\_exceptions=True)  
print(results)[ValueError('Foo value error'), 'Bar finished']

The key parameter here is return\_exceptions. When this parameter is set to True, the function returns a list of both values and exceptions; allowing all tasks to complete before returning. As all tasks have been completed they are removed from the event loop.

If return\_exceptions is set to False (the default value) then asyncio.gather returns immediately, throwing the exception into the calling coroutine. Any task which has not been terminated will remain on the event loop.

The following code explains the behaviour more clearly.

import asyncio

async def foo():

raise ValueError("Foo value error.")

return("Foo finished.")

async def bar():

try:

await asyncio.sleep(1)

return("Bar finished.")

except asyncio.CancelledError:

print("Bar Cancelled.")

async def main():

try:

results = await asyncio.gather(

foo(), bar(),

return\_exceptions=False

)

print(results)

except ValueError as e:

print("Value Error raised.")

asyncio.run(main())

When this code executes foo immediately throws a ValueError exception. This propagates immediately to main where it is caught and an error message printed. As the gather function exits the await statement, as a result of the exception, any pending tasks are cancelled (task.cancel()). This causes a CancelledError to be raised in bar.

Value Error raised.  
Bar Cancelled.

**Always Plan for Cancelled Tasks**

The previous example draws attention to potential cancellation of tasks mid-execution. When planning how the code executes it is quite possible that coroutines are cancelled when an execution is thrown elsewhere in the code. In the case of gather, this may cause a CancelledError to be thrown into a peer coroutine.

If the coroutine in which a CancelledError is raised is communicating with another system, accessing a database or changing resources it may leave the system in an unstable state. Catching the CancelledError provides an opportunity to unwind transactions minimizing the risk of data corruption, inefficiency, or system failure.

Python asyncio is a powerful tool for improving system efficiency using asynchronous execution of code. However, it is very easy to implement code that does not explicitly handle the lifecycle of tasks and exceptions. Planning in advance how tasks are created and cancelled, and how exceptions should be handled, will significantly reduce the amount of time trying to diagnose bugs that may arise.

**The following information is from the docs. It is worth noting the deprecations and changes pertaining to Python v3.10(current) and v3.11(fall)**

[**https://docs.python.org/3/library/asyncio-task.html#asyncio-example-wait-coroutine**](https://docs.python.org/3/library/asyncio-task.html#asyncio-example-wait-coroutine)

[**Waiting Primitives**](https://docs.python.org/3/library/asyncio-task.html#id9)

*coroutine* asyncio.wait(*aws*, *\**, *timeout=None*, *return\_when=ALL\_COMPLETED*)

Run [awaitable objects](https://docs.python.org/3/library/asyncio-task.html#asyncio-awaitables) 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](https://docs.python.org/3/library/asyncio-exceptions.html#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()](https://docs.python.org/3/library/asyncio-task.html#asyncio.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](https://docs.python.org/3/library/asyncio-task.html#asyncio-example-wait-coroutine).

Changed in version 3.10: Removed 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.

Changed in version 3.10: Removed the *loop* parameter.