18.5.9. Develop with asyncio

Asynchronous programming is different than classical "sequential" programming. This page lists common traps and explains how to avoid them.

18.5.9.1. Debug mode of asyncio

The implementation of `asyncio` has been written for performance. In order to ease the development of asynchronous code, you may wish to enable *debug mode*.

To enable all debug checks for an application:

- Enable the asyncio debug mode globally by setting the environment variable `PYTHONASYNCIODEBUG` to `1`, or by calling `AbstractEventLoop.set_debug()`.
- Set the log level of the asyncio logger to `logging.DEBUG`. For example, call `logging.basicConfig(level=logging.DEBUG)` at startup.
- Configure the `warnings` module to display `ResourceWarning` warnings. For example, use the `-Wdefault` command line option of Python to display them.

Examples debug checks:

- Log coroutines defined but never "yielded from"
- `call_soon()` and `call_at()` methods raise an exception if they are called from the wrong thread.
- Log the execution time of the selector
- Log callbacks taking more than 100 ms to be executed. The `AbstractEventLoop.slow_callback_duration` attribute is the minimum duration in seconds of "slow" callbacks.
- `ResourceWarning` warnings are emitted when transports and event loops are not closed explicitly.

See also: The `AbstractEventLoop.set_debug()` method and the asyncio logger.

18.5.9.2. Cancellation

Cancellation of tasks is not common in classic programming. In asynchronous programming, not only is it something common, but you have to prepare your code to handle it.

Futures and tasks can be cancelled explicitly with their `Future.cancel()` method. The `wait_for()` function cancels the waited task when the timeout occurs. There are many other cases where a task can be cancelled indirectly.

Don't call `set_result()` or `set_exception()` method of `Future` if the future is cancelled: it would fail with an exception. For example, write:

```
if not fut.cancelled():
    fut.set_result('done')
```

Don't schedule directly a call to the `set_result()` or the `set_exception()` method of a future with `AbstractEventLoop.call_soon()`: the future can be cancelled before its method is called.

If you wait for a future, you should check early if the future was cancelled to avoid useless operations. Example:

```
@coroutine
def slow_operation(fut):
    if fut.cancelled():
        return
    # ... slow computation ...
    yield from fut
    # ...
```

The `shield()` function can also be used to ignore cancellation.

18.5.9.3. Concurrency and multithreading

An event loop runs in a thread and executes all callbacks and tasks in the same thread. While a task is running in the event loop, no other task is running in the same thread. But when the task uses `yield from`, the task is suspended and the event loop executes the next task.

To schedule a callback from a different thread, the `AbstractEventLoop.call_soon_threadsafe()` method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Most asyncio objects are not thread safe. You should only worry if you access objects outside the event loop. For example, to cancel a future, don't call directly its `Future.cancel()` method, but:

```
loop.call_soon_threadsafe(fut.cancel)
```

To handle signals and to execute subprocesses, the event loop must be run in the main thread.

To schedule a coroutine object from a different thread, the `run_coroutine_threadsafe()` function should be used. It returns a `concurrent.futures.Future` to access the result:

```
future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
result = future.result(timeout)  # Wait for the result with a timeout
```

The `AbstractEventLoop.run_in_executor()` method can be used with a thread pool executor to execute a callback in different thread to not block the thread of the event loop.

See also: The Synchronization primitives section describes ways to synchronize tasks.

The Subprocess and threads section lists asyncio limitations to run subprocesses from different threads.

18.5.9.4. Handle blocking functions correctly

Blocking functions should not be called directly. For example, if a function blocks for 1 second, other tasks are delayed by 1 second which can have an important impact on reactivity.

For networking and subprocesses, the `asyncio` module provides high-level APIs like protocols.

An executor can be used to run a task in a different thread or even in a different process, to not block the thread of the event loop. See the `AbstractEventLoop.run_in_executor()` method.

See also: The Delayed calls section details how the event loop handles time.

18.5.9.5. Logging

The `asyncio` module logs information with the `logging` module in the logger `'asyncio'`.

The default log level for the `asyncio` module is `logging.INFO`. For those not wanting such verbosity from `asyncio` the log level can be changed. For example, to change the level to `logging.WARNING`:

```
logging.getLogger('asyncio').setLevel(logging.WARNING)
```

18.5.9.6. Detect coroutine objects never scheduled

When a coroutine function is called and its result is not passed to `ensure_future()` or to the `AbstractEventLoop.create_task()` method, the execution of the coroutine object will never be scheduled which is probably a bug. Enable the debug mode of asyncio to log a warning to detect it.

Example with the bug:

```
import asyncio

@asyncio.coroutine
def test():
    print("never scheduled")

test()
```

Output in debug mode:

```
Coroutine test() at test.py:3 was never yielded from
Coroutine object created at (most recent call last):
  File "test.py", line 7, in <module>
    test()
```

The fix is to call the `ensure_future()` function or the `AbstractEventLoop.create_task()` method with the coroutine object.

See also: Pending task destroyed.

18.5.9.7. Detect exceptions never consumed

Python usually calls `sys.excepthook()` on unhandled exceptions. If `Future.set_exception()` is called, but the exception is never consumed, `sys.excepthook()` is not called. Instead, a log is emitted when the future is deleted by the garbage collector, with the traceback where the exception was raised.

Example of unhandled exception:

```
import asyncio

@asyncio.coroutine
def bug():
    raise Exception("not consumed")

loop = asyncio.get_event_loop()
```

```
asyncio.ensure_future(bug())
loop.run_forever()
loop.close()
```

Output:

```
Task exception was never retrieved
future: <Task finished coro=<coro() done, defined at asyncio/coroutine
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode of asyncio to get the traceback where the task was created. Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3> excepti
source_traceback: Object created at (most recent call last):
  File "test.py", line 8, in <module>
    asyncio.ensure_future(bug())
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 79, in __next__
    return next(self.gen)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

There are different options to fix this issue. The first option is to chain the coroutine in another coroutine and use classic try/except:

```
@asyncio.coroutine
def handle_exception():
    try:
        yield from bug()
    except Exception:
        print("exception consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(handle_exception())
```

```
loop.run_forever()
loop.close()
```

Another option is to use the `AbstractEventLoop.run_until_complete()` function:

```
task = asyncio.ensure_future(bug())
try:
    loop.run_until_complete(task)
except Exception:
    print("exception consumed")
```

See also: The `Future.exception()` method.

18.5.9.8. Chain coroutines correctly

When a coroutine function calls other coroutine functions and tasks, they should be chained explicitly with `yield from`. Otherwise, the execution is not guaranteed to be sequential.

Example with different bugs using `asyncio.sleep()` to simulate slow operations:

```
import asyncio

@asyncio.coroutine
def create():
    yield from asyncio.sleep(3.0)
    print("(1) create file")

@asyncio.coroutine
def write():
    yield from asyncio.sleep(1.0)
    print("(2) write into file")

@asyncio.coroutine
def close():
    print("(3) close file")

@asyncio.coroutine
def test():
    asyncio.ensure_future(create())
    asyncio.ensure_future(write())
    asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()

loop = asyncio.get_event_loop()
asyncio.ensure_future(test())
loop.run_forever()
```

```
print("Pending tasks at exit: %s" % asyncio.Task.all_tasks(loop))
loop.close()
```

Expected output:

```
(1) create file
(2) write into file
(3) close file
Pending tasks at exit: set()
```

Actual output:

```
(3) close file
(2) write into file
Pending tasks at exit: {<Task pending create() at test.py:7 wait_for=<
Task was destroyed but it is pending!
task: <Task pending create() done at test.py:5 wait_for=<Future pendin
```

The loop stopped before the `create()` finished, `close()` has been called before `write()`, whereas coroutine functions were called in this order: `create()`, `write()`, `close()`.

To fix the example, tasks must be marked with `yield from`:

```
@asyncio.coroutine
def test():
    yield from asyncio.ensure_future(create())
    yield from asyncio.ensure_future(write())
    yield from asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()
```

Or without `asyncio.ensure_future()`:

```
@asyncio.coroutine
def test():
    yield from create()
    yield from write()
    yield from close()
    yield from asyncio.sleep(2.0)
    loop.stop()
```

18.5.9.9. Pending task destroyed

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

Example of log:

```
Task was destroyed but it is pending!
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_f
```

Enable the debug mode of asyncio to get the traceback where the task was created. Example of log in debug mode:

```
Task was destroyed but it is pending!
source_traceback: Object created at (most recent call last):
  File "test.py", line 15, in <module>
    task = asyncio.ensure_future(coro, loop=loop)
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_f
```

See also: Detect coroutine objects never scheduled.

18.5.9.10. Close transports and event loops

When a transport is no more needed, call its `close()` method to release resources. Event loops must also be closed explicitly.

If a transport or an event loop is not closed explicitly, a `ResourceWarning` warning will be emitted in its destructor. By default, `ResourceWarning` warnings are ignored. The Debug mode of asyncio section explains how to display them.