asyncio-eventloop

- Event loop
  - Event loop methods
    - Running and stopping the loop
    - Scheduling callbacks
    - Scheduling delayed callbacks
    - Creating futures and tasks
    - Opening network connections
    - Creating network servers
    - Transferring files
    - TLS upgrade
    - Watching file descriptors
    - Working with socket objects directly
    - DNS
    - Working with pipes
    - Unix signals
    - Executing code in thread or process pools
    - Error handling API
    - Enabling debug mode
    - Running subprocesses
  - Callback handles
  - Server objects
  - Event loop implementations
  - Examples
    - Hello World with call_soon()
    - Display the current date with call_later()
    - Watch a file descriptor for read events
    - Set signal handlers for SIGINT and SIGTERM

Event loop

Source code: `Lib/asyncio/events.py`, `Lib/asyncio/base_events.py`

Preface

The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

Application developers should typically use the high-level asyncio functions, such as `asyncio.run`, and should rarely need to reference the loop object or call its methods. This section is intended mostly for authors of lower-level code, libraries, and frameworks, who need finer control over the event loop behavior.

Obtaining the Event Loop

The following low-level functions can be used to get, set, or create an event loop:

get_running_loop()

Return the running event loop in the current OS thread.

Raise a `RuntimeError` if there is no running event loop.

This function can only be called from a coroutine or a callback.

3.7

get_event_loop()

Get the current event loop.

When called from a coroutine or a callback (e.g. scheduled with call_soon or similar API), this function will always return the running event loop.

If there is no running event loop set, the function will return the loop set by `set_event_loop`, or raise a `RuntimeError` if no loop has been set.

Because this function has rather complex behavior, using the `get_running_loop` function is preferred to `get_event_loop` in coroutines and callbacks.

As noted above, consider using the higher-level `asyncio.run` function, instead of using these lower level functions to manually create and close an event loop.

3.14 Raises a `RuntimeError` if there is no current event loop.

set_event_loop(loop)

Set *loop* as the current event loop for the current OS thread.

new_event_loop()

Create and return a new event loop object.

Contents

This documentation page contains the following sections:

- The Event Loop Methods section is the reference documentation of the event loop APIs;
- The Callback Handles section documents the `Handle` and `TimerHandle` instances which are returned from scheduling methods such as `loop.call_soon` and `loop.call_later`;
- The Server Objects section documents types returned from event loop methods like `loop.create_server`;
- The Event Loop Implementations section documents the `SelectorEventLoop` and `ProactorEventLoop` classes;
- The Examples section showcases how to work with some event loop APIs.

Event loop methods

Event loops have **low-level** APIs for the following:

Running and stopping the loop

loop.run_until_complete(future)

Run until the *future* (an instance of `Future`) has completed.

If the argument is a `coroutine object <coroutine>` it is implicitly scheduled to run as a `asyncio.Task`.

Return the Future's result or raise its exception.

loop.run_forever()

Run the event loop until `stop` is called.

If `stop` is called before `run_forever` is called, the loop will poll the I/O selector once with a timeout of zero, run all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exit.

If `stop` is called while `run_forever` is running, the loop will run the current batch of callbacks and then exit. Note that new callbacks scheduled by callbacks will not run in this case; instead, they will run the next time `run_forever` or `run_until_complete` is called.

loop.stop()

Stop the event loop.

loop.is_running()

Return `True` if the event loop is currently running.

loop.is_closed()

Return `True` if the event loop was closed.

loop.close()

Close the event loop.

The loop must not be running when this function is called. Any pending callbacks will be discarded.

This method clears all queues and shuts down the executor, but does not wait for the executor to finish.

This method is idempotent and irreversible. No other methods should be called after the event loop is closed.

loop.shutdown_asyncgens()

Schedule all currently open `asynchronous generator` objects to close with an `~agen.aclose` call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

Note that there is no need to call this function when `asyncio.run` is used.

Example:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

3.6

loop.shutdown_default_executor(timeout=None)

Schedule the closure of the default executor and wait for it to join all of the threads in the `~concurrent.futures.ThreadPoolExecutor`. Once this method has been called, using the default executor with `loop.run_in_executor` will raise a `RuntimeError`.

The *timeout* parameter specifies the amount of time (in `float` seconds) the executor will be given to finish joining. With the default, `None`, the executor is allowed an unlimited amount of time.

If the *timeout* is reached, a `RuntimeWarning` is emitted and the default executor is terminated without waiting for its threads to finish joining.

> [!NOTE] Do not call this method when using `asyncio.run`, as the latter handles default executor shutdown automatically.

3.9

3.12 Added the *timeout* parameter.

Scheduling callbacks

loop.call_soon(callback, *args, context=None)

Schedule the *callback* `callback` to be called with *args* arguments at the next iteration of the event loop.

Return an instance of `asyncio.Handle`, which can be used later to cancel the callback.

Callbacks are called in the order in which they are registered. Each callback will be called exactly once.

The optional keyword-only *context* argument specifies a custom `contextvars.Context` for the *callback* to run in. Callbacks use the current context when no *context* is provided.

Unlike `call_soon_threadsafe`, this method is not thread-safe.

loop.call_soon_threadsafe(callback, *args, context=None)

A thread-safe variant of `call_soon`. When scheduling callbacks from another thread, this function *must* be used, since `call_soon` is not thread-safe.

This function is safe to be called from a reentrant context or signal handler, however, it is not safe or fruitful to use the returned handle in such contexts.

Raises `RuntimeError` if called on a loop that's been closed. This can happen on a secondary thread when the main application is shutting down.

See the `concurrency and multithreading <asyncio-multithreading>` section of the documentation.

3.7 The *context* keyword-only parameter was added. See `567` for more details.

> [!NOTE] Most `asyncio` scheduling functions don't allow passing keyword arguments. To do that, use `functools.partial`:
>
> ```
> # will schedule "print("Hello", flush=True)"
> loop.call_soon(
>     functools.partial(print, "Hello", flush=True))
> ```
>
> Using partial objects is usually more convenient than using lambdas, as asyncio can render partial objects better in debug and error messages.

Scheduling delayed callbacks

Event loop provides mechanisms to schedule callback functions to be called at some point in the future. Event loop uses monotonic clocks to track time.

loop.call_later(delay, callback, *args, context=None)

Schedule *callback* to be called after the given *delay* number of seconds (can be either an int or a float).

An instance of `asyncio.TimerHandle` is returned which can be used to cancel the callback.

callback will be called exactly once. If two callbacks are scheduled for exactly the same time, the order in which they are called is undefined.

The optional positional *args* will be passed to the callback when it is called. Use `functools.partial to pass keyword arguments <asyncio-pass-keywords>` to *callback*.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *callback* to run in. The current context is used when no *context* is provided.

> [!NOTE] For performance, callbacks scheduled with `loop.call_later` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

3.7 The *context* keyword-only parameter was added. See `567` for more details.

3.8 In Python 3.7 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.8.

loop.call_at(when, callback, *args, context=None)

Schedule *callback* to be called at the given absolute timestamp *when* (an int or a float), using the same time reference as `loop.time`.

This method's behavior is the same as `call_later`.

An instance of `asyncio.TimerHandle` is returned which can be used to cancel the callback.

> [!NOTE] For performance, callbacks scheduled with `loop.call_at` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

3.7 The *context* keyword-only parameter was added. See `567` for more details.

3.8 In Python 3.7 and earlier with the default event loop implementation, the difference between *when* and the current time could not exceed one day. This has been fixed in Python 3.8.

loop.time()

Return the current time, as a `float` value, according to the event loop's internal monotonic clock.

> [!NOTE] .. versionchanged:: 3.8 In Python 3.7 and earlier timeouts (relative *delay* or absolute *when*) should not exceed one day. This has been fixed in Python 3.8.

The `asyncio.sleep` function.

Creating futures and tasks

loop.create_future()

Create an `asyncio.Future` object attached to the event loop.

This is the preferred way to create Futures in asyncio. This lets third-party event loops provide alternative implementations of the Future object (with better performance or instrumentation).

3.5.2

loop.create_task(coro, , *name=None, context=None, eager_start=None,*kwargs)

Schedule the execution of `coroutine <coroutine>` *coro*. Return a `Task` object.

Third-party event loops can use their own subclass of `Task` for interoperability. In this case, the result type is a subclass of `Task`.

The full function signature is largely the same as that of the `Task` constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

If the *name* argument is provided and not `None`, it is set as the name of the task using `Task.set_name`.

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

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by `loop.set_task_factory` will be used.

3.8 Added the *name* parameter.

3.11 Added the *context* parameter.

3.13.3 Added `kwargs` which passes on arbitrary extra parameters, including `name` and `context`.

3.13.4 Rolled back the change that passes on *name* and *context* (if it is None), while still passing on other arbitrary keyword arguments (to avoid breaking backwards compatibility with 3.13.3).

3.14 All *kwargs* are now passed on. The *eager_start* parameter works with eager task factories.

loop.set_task_factory(factory)

Set a task factory that will be used by `loop.create_task`.

If *factory* is `None` the default task factory will be set. Otherwise, *factory* must be a *callable* with the signature matching `(loop, coro, **kwargs)`, where *loop* is a reference to the active event loop, and *coro* is a coroutine object. The callable must pass on all *kwargs*, and return a `asyncio.Task`-compatible object.

3.13.3 Required that all *kwargs* are passed on to `asyncio.Task`.

3.13.4 *name* is no longer passed to task factories. *context* is no longer passed to task factories if it is `None`.

3.14 *name* and *context* are now unconditionally passed on to task factories again.

loop.get_task_factory()

Return a task factory or `None` if the default one is in use.

Opening network connections

loop.create_connection(protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, happy_eyeballs_delay=None, interleave=None, all_errors=False)

Open a streaming transport connection to a given address specified by *host* and *port*.

The socket family can be either `~socket.AF_INET` or `~socket.AF_INET6` depending on *host* (or the *family* argument, if provided).

The socket type will be `~socket.SOCK_STREAM`.

protocol_factory must be a callable returning an `asyncio protocol <asyncio-protocol>` implementation.

This method will try to establish the connection in the background. When successful, it returns a `(transport, protocol)` pair.

The chronological synopsis of the underlying operation is as follows:

1. The connection is established and a `transport <asyncio-transport>` is created for it.
2. *protocol_factory* is called without arguments and is expected to return a `protocol <asyncio-protocol>` instance.

3. The protocol instance is coupled with the transport by calling its `~BaseProtocol.connection_made` method.
4. A `(transport, protocol)` tuple is returned on success.

The created transport is an implementation-dependent bidirectional stream.

Other arguments:

- *ssl*: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a `ssl.SSLContext` object, this context is used to create the transport; if *ssl* is `True`, a default context returned from `ssl.create_default_context` is used. `SSL/TLS security considerations <ssl-security>`
- *server_hostname* sets or overrides the hostname that the target server's certificate will be matched against. Should only be passed if *ssl* is not `None`. By default the value of the *host* argument is used. If *host* is empty, there is no default and you must pass a value for *server_hostname*. If *server_hostname* is an empty string, hostname matching is disabled (which is a serious security risk, allowing for potential man-in-the-middle attacks).
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to getaddrinfo() for *host* resolution. If given, these should all be integers from the corresponding `socket` module constants.
- *happy_eyeballs_delay*, if given, enables Happy Eyeballs for this connection. It should be a floating-point number representing the amount of time in seconds to wait for a connection attempt to complete, before starting the next attempt in parallel. This is the "Connection Attempt Delay" as defined in `8305`. A sensible default value recommended by the RFC is `0.25` (250 milliseconds).
- *interleave* controls address reordering when a host name resolves to multiple IP addresses. If `0` or unspecified, no reordering is done, and addresses are tried in the order returned by `getaddrinfo`. If a positive integer is specified, the addresses are interleaved by address family, and the given integer is interpreted as "First Address Family Count" as defined in `8305`. The default is `0` if *happy_eyeballs_delay* is not specified, and `1` if it is.
- *sock*, if given, should be an existing, already connected `socket.socket` object to be used by the transport. If *sock* is given, none of *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* and *local_addr* should be specified.

  > [!NOTE] The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `~asyncio.BaseTransport.close` method.

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port* .
- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

- *all_errors* determines what exceptions are raised when a connection cannot be created. By default, only a single `Exception` is raised: the first exception if there is only one or all errors have same message, or a single `OSError` with the error messages combined. When `all_errors` is `True`, an `ExceptionGroup` will be raised containing all exceptions (even if there is only one).

3.5

Added support for SSL/TLS in `ProactorEventLoop`.

3.6

The socket option `socket.TCP_NODELAY <socket-unix-constants>` is set by default for all TCP connections.

3.7

Added the *ssl_handshake_timeout* parameter.

3.8

Added the *happy_eyeballs_delay* and *interleave* parameters.

Happy Eyeballs Algorithm: Success with Dual-Stack Hosts. When a server's IPv4 path and protocol are working, but the server's IPv6 path and protocol are not working, a dual-stack client application experiences significant connection delay compared to an IPv4-only client. This is undesirable because it causes the dual-stack client to have a worse user experience. This document specifies requirements for algorithms that reduce this user-visible delay and provides an algorithm.

For more information: https://datatracker.ietf.org/doc/html/rfc6555

3.11

Added the *ssl_shutdown_timeout* parameter.

3.12 *all_errors* was added.

3.16 Raises a `ValueError` if `ssl.check_hostname` is `True` and `server_hostname` is not supplied.

The `open_connection` function is a high-level alternative API. It returns a pair of (`StreamReader`, `StreamWriter`) that can be used directly in async/await code.

loop.create_datagram_endpoint(protocol_factory, local_addr=None, remote_addr=None, *, family=0, proto=0, flags=0, reuse_port=None, allow_broadcast=None, sock=None)

Create a datagram connection.

The socket family can be either `~socket.AF_INET`, `~socket.AF_INET6`, or `~socket.AF_UNIX`, depending on *host* (or the *family* argument, if provided).

The socket type will be `~socket.SOCK_DGRAM`.

protocol_factory must be a callable returning a `protocol <asyncio-protocol>` implementation.

A tuple of `(transport, protocol)` is returned on success.

Other arguments:

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using `getaddrinfo`.

  > [!NOTE] On Windows, when using the proactor event loop with `local_addr=None`, an `OSError` with `!errno.WSAEINVAL` will be raised when running it.

- *remote_addr*, if given, is a `(remote_host, remote_port)` tuple used to connect the socket to a remote address. The *remote_host* and *remote_port* are looked up using `getaddrinfo`.
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to `getaddrinfo` for *host* resolution. If given, these should all be integers from the corresponding `socket` module constants.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some Unixes. If the `socket.SO_REUSEPORT <socket-unix-constants>` constant is not defined then this capability is unsupported.
- *allow_broadcast* tells the kernel to allow this endpoint to send messages to the broadcast address.
- *sock* can optionally be specified in order to use a preexisting, already connected, `socket.socket` object to be used by the transport. If specified, *local_addr* and *remote_addr* should be omitted (must be `None`).

  > [!NOTE] The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `~asyncio.BaseTransport.close` method.

See `UDP echo client protocol <asyncio-udp-echo-client-protocol>` and `UDP echo server protocol <asyncio-udp-echo-server-protocol>` examples.

3.4.4 The *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast*, and *sock* parameters were added.

3.8 Added support for Windows.

3.8.1 The *reuse_address* parameter is no longer supported, as using `socket.SO_REUSEADDR <socket-unix-constants>` poses a significant security concern for UDP. Explicitly passing `reuse_address=True` will raise an exception.

When multiple processes with differing UIDs assign sockets to an identical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, `socket.SO_REUSEPORT <socket-unix-constants>` is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.

3.11 The *reuse_address* parameter, disabled since Python 3.8.1, 3.7.6 and 3.6.10, has been entirely removed.

loop.create_unix_connection(protocol_factory, path=None, *, ssl=None, sock=None, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None)

Create a Unix connection.

The socket family will be `~socket.AF_UNIX`; socket type will be `~socket.SOCK_STREAM`.

A tuple of `(transport, protocol)` is returned on success.

path is the name of a Unix domain socket and is required, unless a *sock* parameter is specified. Abstract Unix sockets, `str`, `bytes`, and `~pathlib.Path` paths are supported.

See the documentation of the `loop.create_connection` method for information about arguments to this method.

Unix.

3.7 Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a `path-like object`.

3.11

Added the *ssl_shutdown_timeout* parameter.

Creating network servers

loop.create_server(protocol_factory, host=None, port=None, *, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, keep_alive=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, start_serving=True)

Create a TCP server (socket type `~socket.SOCK_STREAM`) listening on *port* of the *host* address.

Returns a `Server` object.

Arguments:

- *protocol_factory* must be a callable returning a `protocol <asyncio-protocol>` implementation.
- The *host* parameter can be set to several types which determine where the server would be listening:
  - If *host* is a string, the TCP server is bound to a single network interface specified by *host*.
  - If *host* is a sequence of strings, the TCP server is bound to all network interfaces specified by the sequence.
  - If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).
- *family* can be set to either `socket.AF_INET` or `~socket.AF_INET6` to force the socket to use IPv4 or IPv6. If not set, the *family* will be determined from host name (defaults to `~socket.AF_UNSPEC`).
- *flags* is a bitmask for `getaddrinfo`.
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* must not be specified.

  > [!NOTE] The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `~asyncio.Server.close` method.

- *backlog* is the maximum number of queued connections passed to `~socket.socket.listen` (defaults to 100).
- *ssl* can be set to an `~ssl.SSLContext` instance to enable TLS over the accepted connections.
- *reuse_address* tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on Unix.
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.
- *keep_alive* set to `True` keeps connections active by enabling the periodic transmission of messages.

3.13

Added the *keep_alive* parameter.

- *ssl_handshake_timeout* is (for a TLS server) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).
- *start_serving* set to `True` (the default) causes the created server to start accepting connections immediately. When set to `False`, the user should await on `Server.start_serving` or `Server.serve_forever` to make the server to start accepting connections.

3.5

Added support for SSL/TLS in `ProactorEventLoop`.

3.5.1

The *host* parameter can be a sequence of strings.

3.6

Added *ssl_handshake_timeout* and *start_serving* parameters. The socket option `socket.TCP_NODELAY <socket-unix-constants>` is set by default for all TCP connections.

3.11

Added the *ssl_shutdown_timeout* parameter.

The `start_server` function is a higher-level alternative API that returns a pair of `StreamReader` and `StreamWriter` that can be used in an async/await code.

loop.create_unix_server(protocol_factory, path=None, *, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, start_serving=True, cleanup_socket=True, mode=None)

Similar to `loop.create_server` but works with the `~socket.AF_UNIX` socket family.

path is the name of a Unix domain socket, and is required, unless a *sock* argument is provided. Abstract Unix sockets, `str`, `bytes`, and `~pathlib.Path` paths are supported.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

If *mode* is not `None`, the permissions of the socket file created for *path* are changed to *mode* (as accepted by `os.chmod`) right after binding, before the server starts accepting connections, so a connection can never be accepted while the default, umask-derived permissions are still in effect. *mode* cannot be combined with *sock* and is not supported for abstract Unix sockets.

See the documentation of the `loop.create_server` method for information about arguments to this method.

Unix.

3.7

Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a `~pathlib.Path` object.

3.11

Added the *ssl_shutdown_timeout* parameter.

3.13

Added the *cleanup_socket* parameter.

3.16

Added the *mode* parameter.

loop.connect_accepted_socket(protocol_factory, sock, *, ssl=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None)

Wrap an already accepted connection into a transport/protocol pair.

This method can be used by servers that accept connections outside of asyncio but that use asyncio to handle them.

Parameters:

- *protocol_factory* must be a callable returning a `protocol <asyncio-protocol>` implementation.
- *sock* is a preexisting socket object returned from `socket.accept <socket.socket.accept>`.

  > [!NOTE] The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `~asyncio.BaseTransport.close` method.

- *ssl* can be set to an `~ssl.SSLContext` to enable SSL over the accepted connections.
- *ssl_handshake_timeout* is (for an SSL connection) the time in seconds to wait for the SSL handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Returns a `(transport, protocol)` pair.

3.5.3

3.7

Added the *ssl_handshake_timeout* parameter.

3.11

Added the *ssl_shutdown_timeout* parameter.

Transferring files

loop.sendfile(transport, file, offset=0, count=None, *, fallback=True)

Send a *file* over a *transport*. Return the total number of bytes sent.

The method uses high-performance `os.sendfile` if available.

file must be a regular file object opened in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and `file.tell() <io.IOBase.tell>` can be used to obtain the actual number of bytes sent.

fallback set to `True` makes asyncio to manually read and send the file when the platform does not support the sendfile system call (e.g. Windows or SSL socket on Unix).

Raise `SendfileNotAvailableError` if the system does not support the *sendfile* syscall and *fallback* is `False`.

3.7

TLS upgrade

loop.start_tls(transport, protocol, sslcontext, *, server_side=False, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None)

Upgrade an existing transport-based connection to TLS.

Create a TLS coder/decoder instance and insert it between the *transport* and the *protocol*. The coder/decoder implements both *transport*-facing protocol and *protocol*-facing transport.

Return the created two-interface instance. After *await*, the *protocol* must stop using the original *transport* and communicate with the returned object only because the coder caches *protocol*-side data and sporadically exchanges extra TLS session packets with *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

Parameters:

- *transport* and *protocol* instances that methods like `~loop.create_server` and `~loop.create_connection` return.
- *sslcontext*: a configured instance of `~ssl.SSLContext`.
- *server_side* pass `True` when a server-side connection is being upgraded (like the one created by `~loop.create_server`).
- *server_hostname*: sets or overrides the host name that the target server's certificate will be matched against.
- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

3.7

3.11

Added the *ssl_shutdown_timeout* parameter.

Watching file descriptors

loop.add_reader(fd, callback, *args)

Start monitoring the *fd* file descriptor for read availability and invoke *callback* with the specified arguments once *fd* is available for reading.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

loop.remove_reader(fd)

Stop monitoring the *fd* file descriptor for read availability. Returns `True` if *fd* was previously being monitored for reads.

loop.add_writer(fd, callback, *args)

Start monitoring the *fd* file descriptor for write availability and invoke *callback* with the specified arguments *args* once *fd* is available for writing.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

Use `functools.partial` `to pass keyword arguments <asyncio-pass-keywords>` to *callback*.

loop.remove_writer(fd)

Stop monitoring the *fd* file descriptor for write availability. Returns `True` if *fd* was previously being monitored for writes.

See also `Platform Support <asyncio-platform-support>` section for some limitations of these methods.

Working with socket objects directly

In general, protocol implementations that use transport-based APIs such as `loop.create_connection` and `loop.create_server` are faster than implementations that work with sockets directly. However, there are some use cases when performance is not critical, and working with `~socket.socket` objects directly is more convenient.

loop.sock_recv(sock, nbytes)

Receive up to *nbytes* from *sock*. Asynchronous version of `socket.recv() <socket.socket.recv>`.

Return the received data as a bytes object.

sock must be a non-blocking socket.

3.7 Even though this method was always documented as a coroutine method, releases before Python 3.7 returned a `Future`. Since Python 3.7 this is an `async def` method.

loop.sock_recv_into(sock, buf)

Receive data from *sock* into the *buf* buffer. Modeled after the blocking `socket.recv_into() <socket.socket.recv_into>` method.

Return the number of bytes written to the buffer.

sock must be a non-blocking socket.

3.7

loop.sock_recvfrom(sock, bufsize)

Receive a datagram of up to *bufsize* from *sock*. Asynchronous version of `socket.recvfrom() <socket.socket.recvfrom>`.

Return a tuple of (received data, remote address).

sock must be a non-blocking socket.

3.11

loop.sock_recvfrom_into(sock, buf, nbytes=0)

Receive a datagram of up to *nbytes* from *sock* into *buf*. Asynchronous version of `socket.recvfrom_into() <socket.socket.recvfrom_into>`.

Return a tuple of (number of bytes received, remote address).

sock must be a non-blocking socket.

3.11

loop.sock_sendall(sock, data)

Send *data* to the *sock* socket. Asynchronous version of `socket.sendall() <socket.socket.sendall>`.

This method continues to send to the socket until either all data in *data* has been sent or an error occurs. `None` is returned on success. On error, an exception is raised. Additionally, there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

sock must be a non-blocking socket.

3.7 Even though the method was always documented as a coroutine method, before Python 3.7 it returned a `Future`. Since Python 3.7, this is an `async def` method.

loop.sock_sendto(sock, data, address)

Send a datagram from *sock* to *address*. Asynchronous version of `socket.sendto() <socket.socket.sendto>`.

Return the number of bytes sent.

sock must be a non-blocking socket.

3.11

loop.sock_connect(sock, address)

Connect *sock* to a remote socket at *address*.

Asynchronous version of `socket.connect() <socket.socket.connect>`.

sock must be a non-blocking socket.

With `SelectorEventLoop`, *address* does not need to be resolved: for `~socket.AF_INET` and `~socket.AF_INET6` sockets, `sock_connect` first checks whether *address* is already resolved by calling `socket.inet_pton`, and uses `loop.getaddrinfo` to resolve it if it is not.

`ProactorEventLoop`, the default event loop on Windows, does not resolve *address*. The host must already be a numeric IP address; passing a host name raises `OSError`. Resolve the address with `loop.getaddrinfo` first, or use `loop.create_connection`, which resolves the address on every platform.

3.5.2 With `SelectorEventLoop`, `address` no longer needs to be resolved.

`loop.create_connection` and `asyncio.open_connection() <open_connection>`.

loop.sock_accept(sock)

Accept a connection. Modeled after the blocking `socket.accept() <socket.socket.accept>` method.

The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

sock must be a non-blocking socket.

3.7 Even though the method was always documented as a coroutine method, before Python 3.7 it returned a `Future`. Since Python 3.7, this is an `async def` method.

`loop.create_server` and `start_server`.

loop.sock_sendfile(sock, file, offset=0, count=None, *, fallback=True)

Send a file using high-performance `os.sendfile` if possible. Return the total number of bytes sent.

Asynchronous version of `socket.sendfile() <socket.socket.sendfile>`.

sock must be a non-blocking `socket.SOCK_STREAM` `~socket.socket`.

file must be a regular file object open in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and `file.tell() <io.IOBase.tell>` can be used to obtain the actual number of bytes sent.

fallback, when set to `True`, makes asyncio manually read and send the file when the platform does not support the sendfile syscall (e.g. Windows or SSL socket on Unix).

Raise `SendfileNotAvailableError` if the system does not support *sendfile* syscall and *fallback* is `False`.

sock must be a non-blocking socket.

3.7

DNS

loop.getaddrinfo(host, port, *, family=0, type=0, proto=0, flags=0)

Asynchronous version of `socket.getaddrinfo`.

loop.getnameinfo(sockaddr, flags=0)

Asynchronous version of `socket.getnameinfo`.

> [!NOTE] Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop's default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

3.7 Both *getaddrinfo* and *getnameinfo* methods were always documented to return a coroutine, but prior to Python 3.7 they were, in fact, returning `asyncio.Future` objects. Starting with Python 3.7 both methods are coroutines.

Working with pipes

loop.connect_read_pipe(protocol_factory, pipe)

Register the read end of *pipe* in the event loop.

protocol_factory must be a callable returning an `asyncio protocol <asyncio-protocol>` implementation.

pipe is a `file-like object <file object>`. See `Supported pipe objects <asyncio-pipe-objects>` for the objects supported as *pipe*.

Return pair `(transport, protocol)`, where *transport* supports the `ReadTransport` interface and *protocol* is an object instantiated by the *protocol_factory*.

With `SelectorEventLoop` event loop, the *pipe* is set to non-blocking mode.

loop.connect_write_pipe(protocol_factory, pipe)

Register the write end of *pipe* in the event loop.

protocol_factory must be a callable returning an `asyncio protocol <asyncio-protocol>` implementation.

pipe is a `file-like object <file object>`. See `Supported pipe objects <asyncio-pipe-objects>` for the objects supported as *pipe*.

Return pair `(transport, protocol)`, where *transport* supports `WriteTransport` interface and *protocol* is an object instantiated by the *protocol_factory*.

With `SelectorEventLoop` event loop, the *pipe* is set to non-blocking mode.

Supported pipe objects

These methods only work with objects the operating system can poll for readiness or perform overlapped I/O on. Regular files on disk are **not** supported on any platform. There is no asynchronous file I/O in asyncio; use `loop.run_in_executor` to read and write regular files without blocking the event loop.

On Unix, with `SelectorEventLoop`, *pipe* must wrap one of the following:

- a pipe, such as an end of an `os.pipe` pair or a FIFO created with `os.mkfifo`;
- a socket;
- a character device, such as a terminal.

On Windows, where only `ProactorEventLoop` implements these methods, *pipe* must wrap a handle opened for overlapped I/O (that is, created with the `FILE_FLAG_OVERLAPPED` flag), since the handle has to be associated with an I/O completion port. Handles that were not opened for overlapped I/O are rejected. In particular, the standard streams (`sys.stdin`, `sys.stdout` and `sys.stderr`), console handles, and the pipes created by `os.pipe` are **not** opened for overlapped I/O and therefore cannot be used with these methods.

> [!NOTE] `SelectorEventLoop` does not support the above methods on Windows. Use `ProactorEventLoop` instead for Windows.

The `loop.subprocess_exec` and `loop.subprocess_shell` methods.

Unix signals

loop.add_signal_handler(signum, callback, *args)

Set *callback* as the handler for the *signum* signal, passing *args* as positional arguments.

The callback will be invoked by *loop*, along with other queued callbacks and runnable coroutines of that event loop. Unlike signal handlers registered using `signal.signal`, a callback registered with this function is allowed to interact with the event loop.

Raise `ValueError` if the signal number is invalid or uncatchable. Raise `RuntimeError` if there is a problem setting up the handler.

Use `functools.partial` `to pass keyword arguments <asyncio-pass-keywords>` to *callback*.

Like `signal.signal`, this function must be invoked in the main thread.

loop.remove_signal_handler(sig)

Remove the handler for the *sig* signal.

Return `True` if the signal handler was removed, or `False` if no handler was set for the given signal.

Unix.

The `signal` module.

Executing code in thread or process pools

loop.run_in_executor(executor, func, *args)

Arrange for *func* to be called in the specified executor passing *args* as positional arguments.

The *executor* argument should be an `concurrent.futures.Executor` instance. The default executor is used if *executor* is `None`. The default executor can be set by `loop.set_default_executor`, otherwise, a `concurrent.futures.ThreadPoolExecutor` will be lazy-initialized and used by `run_in_executor` if needed.

Example:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

    # 4. Run in a custom interpreter pool:
```

```
    with concurrent.futures.InterpreterPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom interpreter pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

Note that the entry point guard (`if __name__ == '__main__'`) is required for option 3 due to the peculiarities of `multiprocessing`, which is used by `~concurrent.futures.ProcessPoolExecutor`. See `Safe importing of main module <multiprocessing-safe-main-import>`.

This method returns a `asyncio.Future` object.

Use `functools.partial` `to pass keyword arguments <asyncio-pass-keywords>` to *func*.

3.5.3 `loop.run_in_executor` no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (`~concurrent.futures.ThreadPoolExecutor`) to set the default.

loop.set_default_executor(executor)

Set *executor* as the default executor used by `run_in_executor`. *executor* must be an instance of `~concurrent.futures.ThreadPoolExecutor`, which includes `~concurrent.futures.InterpreterPoolExecutor`.

3.11 *executor* must be an instance of `~concurrent.futures.ThreadPoolExecutor`.

Error handling API

Allows customizing how exceptions are handled in the event loop.

loop.set_exception_handler(handler)

Set *handler* as the new event loop exception handler.

If *handler* is `None`, the default exception handler will be set. Otherwise, *handler* must be a callable with the signature matching `(loop, context)`, where `loop` is a reference to the active event loop, and `context` is a `dict` object containing the details of the exception (see `call_exception_handler` documentation for details about context).

If the handler is called on behalf of a `~asyncio.Task` or `~asyncio.Handle`, it is run in the `contextvars.Context` of that task or callback handle.

3.12

The handler may be called in the `~contextvars.Context` of the task or handle where the exception originated.

loop.get_exception_handler()

Return the current exception handler, or `None` if no custom exception handler was set.

3.5.2

loop.default_exception_handler(context)

Default exception handler.

This is called when an exception occurs and no exception handler is set. This can be called by a custom exception handler that wants to defer to the default handler behavior.

context parameter has the same meaning as in `call_exception_handler`.

loop.call_exception_handler(context)

Call the current event loop exception handler.

context is a `dict` object containing the following keys (new keys may be introduced in future Python versions):

- 'message': Error message;
- 'exception' (optional): Exception object;
- 'future' (optional): `asyncio.Future` instance;
- 'task' (optional): `asyncio.Task` instance;
- 'handle' (optional): `asyncio.Handle` instance;
- 'protocol' (optional): `Protocol <asyncio-protocol>` instance;
- 'transport' (optional): `Transport <asyncio-transport>` instance;
- 'socket' (optional): `socket.socket` instance;
- 'source_traceback' (optional): Traceback of the source;
- 'handle_traceback' (optional): Traceback of the handle;
- 'asyncgen' (optional): Asynchronous generator that caused the exception.

> [!NOTE] This method should not be overloaded in subclassed event loops. For custom exception handling, use the `set_exception_handler` method.

Enabling debug mode

loop.get_debug()

Get the debug mode (`bool`) of the event loop.

The default value is `True` if the environment variable `PYTHONASYNCIODEBUG` is set to a non-empty string, `False` otherwise.

loop.set_debug(enabled: bool)

Set the debug mode of the event loop.

3.7

The new `Python Development Mode <devmode>` can now also be used to enable the debug mode.

loop.slow_callback_duration

This attribute can be used to set the minimum execution duration in seconds that is considered "slow". When debug mode is enabled, "slow" callbacks are logged.

Default value is 100 milliseconds.

The `debug mode of asyncio <asyncio-debug-mode>`.

Running subprocesses

Methods described in this subsections are low-level. In regular async/await code consider using the high-level `asyncio.create_subprocess_shell` and `asyncio.create_subprocess_exec` convenience functions instead.

> [!NOTE] On Windows, the default event loop `ProactorEventLoop` supports subprocesses, whereas `SelectorEventLoop` does not. See `Subprocess Support on Windows <asyncio-windows-subprocess>` for details.

loop.subprocess_exec(protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE,*kwargs)

Create a subprocess from one or more string arguments specified by *args*.

args must be a list of strings represented by:

- `str`;

- or `bytes`, encoded to the `filesystem encoding <filesystem-encoding>`.

The first string specifies the program executable, and the remaining strings specify the arguments. Together, string arguments form the `argv` of the program.

This is similar to the standard library `subprocess.Popen` class called with `shell=False` and the list of strings passed as the first argument; however, where `~subprocess.Popen` takes a single argument which is list of strings, *subprocess_exec* takes multiple string arguments.

The *protocol_factory* must be a callable returning a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- *stdin* can be any of these:
  - a file-like object
  - an existing file descriptor (a positive integer), for example those created with `os.pipe`
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stdout* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stderr* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
  - the `subprocess.STDOUT` constant which will connect the standard error stream to the process' standard output stream
- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* and *errors*, which should not be specified at all.
  The `asyncio` subprocess API does not support decoding the streams as text. `bytes.decode` can be used to convert the bytes returned from the stream to text.

If a file-like object passed as *stdin*, *stdout* or *stderr* represents a pipe, then the other side of this pipe should be registered with `~loop.connect_write_pipe` or `~loop.connect_read_pipe` for use with the event loop.

See the constructor of the `subprocess.Popen` class for documentation on other arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the `asyncio.SubprocessTransport` base class and *protocol* is an object instantiated by the *protocol_factory*.

If the transport is closed or is garbage collected, the child process is killed if it is still running.

loop.subprocess_shell(protocol_factory, cmd, *, *stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE,**kwargs)

Create a subprocess from *cmd*, which can be a `str` or a `bytes` string encoded to the `filesystem encoding <filesystem-encoding>`, using the platform's "shell" syntax.

This is similar to the standard library `subprocess.Popen` class called with `shell=True`.

The *protocol_factory* must be a callable returning a subclass of the `SubprocessProtocol` class.

See `~loop.subprocess_exec` for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the `SubprocessTransport` base class and *protocol* is an object instantiated by the *protocol_factory*.

If the transport is closed or is garbage collected, the child process is killed if it is still running.

> [!NOTE] It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote` function can be used to properly escape whitespace and special characters in strings that are going to be used to construct shell commands.

Callback handles

A callback wrapper object returned by `loop.call_soon`, `loop.call_soon_threadsafe`.

get_context()

Return the `contextvars.Context` object associated with the handle.

3.12

cancel()

Cancel the callback. If the callback has already been canceled or executed, this method has no effect.

cancelled()

Return `True` if the callback was cancelled.

3.7

A callback wrapper object returned by `loop.call_later`, and `loop.call_at`.

This class is a subclass of `Handle`.

when()

Return a scheduled callback time as `float` seconds.

The time is an absolute timestamp, using the same time reference as `loop.time`.

3.7

Server objects

Server objects are created by `loop.create_server`, `loop.create_unix_server`, `start_server`, and `start_unix_server` functions.

Do not instantiate the `Server` class directly.

Server objects are asynchronous context managers. When used in an `async with` statement, it's guaranteed that the Server object is closed and not accepting new connections when the `async with` statement is completed:

```
srv = await loop.create_server(...)

async with srv:
    # some code
```

```
# At this point, srv is closed and no longer accepts new connections.
```

3.7 Server object is an asynchronous context manager since Python 3.7.

3.11 This class was exposed publicly as `asyncio.Server` in Python 3.9.11, 3.10.3 and 3.11.

close()

Stop serving: close listening sockets and set the `sockets` attribute to `None`.

The sockets that represent existing incoming client connections are left open.

The server is closed asynchronously; use the `wait_closed` coroutine to wait until the server is closed (and no more connections are active).

close_clients()

Close all existing incoming client connections.

Calls `~asyncio.BaseTransport.close` on all associated transports.

`close` should be called before `close_clients` when closing the server to avoid races with new clients connecting.

3.13

abort_clients()

Close all existing incoming client connections immediately, without waiting for pending operations to complete.

Calls `~asyncio.WriteTransport.abort` on all associated transports.

`close` should be called before `abort_clients` when closing the server to avoid races with new clients connecting.

3.13

get_loop()

Return the event loop associated with the server object.

3.7

start_serving()

Start accepting connections.

This method is idempotent, so it can be called when the server is already serving.

The *start_serving* keyword-only parameter to `loop.create_server` and `asyncio.start_server` allows creating a Server object that is not accepting connections initially. In this case `Server.start_serving()`, or `Server.serve_forever` can be used to make the Server start accepting connections.

3.7

serve_forever()

Start accepting connections until the coroutine is cancelled. Cancellation of `serve_forever` task causes the server to be closed.

This method can be called if the server is already accepting connections. Only one `serve_forever` task can exist per one *Server* object.

Example:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

3.7

is_serving()

Return `True` if the server is accepting new connections.

3.7

wait_closed()

Wait until the `close` method completes and all active connections have finished.

3.12 `wait_closed()` now waits until the server is closed and all active connections have finished. Previously, it returned immediately if the server was already closed, even if connections were still active.

sockets

List of socket-like objects, `asyncio.trsock.TransportSocket`, which the server is listening on.

3.7 Prior to Python 3.7 `Server.sockets` used to return an internal list of server sockets directly. In 3.7 a copy of that list is returned.

Event loop implementations

asyncio ships with two different event loop implementations: `SelectorEventLoop` and `ProactorEventLoop`.

By default asyncio is configured to use `EventLoop`.

A subclass of `AbstractEventLoop` based on the `selectors` module.

Uses the most efficient *selector* available for the given platform. It is also possible to manually configure the exact selector implementation to be used:

```
import asyncio
import selectors

async def main():
    ...

loop_factory = lambda: asyncio.SelectorEventLoop(selectors.SelectSelector())
asyncio.run(main(), loop_factory=loop_factory)
```

Unix, Windows.

A subclass of `AbstractEventLoop` for Windows that uses "I/O Completion Ports" (IOCP).

Windows.

MSDN documentation on I/O Completion Ports.

An alias to the most efficient available subclass of `AbstractEventLoop` for the given platform.

It is an alias to `SelectorEventLoop` on Unix and `ProactorEventLoop` on Windows.

3.13

Abstract base class for asyncio-compliant event loops.

The `asyncio-event-loop-methods` section lists all methods that an alternative implementation of `AbstractEventLoop` should have defined.

A base class that partially implements `AbstractEventLoop` to simplify writing custom event loops.

It provides default implementations of many `AbstractEventLoop` methods, so subclasses have fewer methods to implement themselves. In return, a subclass is expected to implement a number of internal, protected methods. See `writing_custom_event_loop` for more information.

Examples

Note that all examples in this section **purposefully** show how to use the low-level event loop APIs, such as `loop.run_forever` and `loop.call_soon`. Modern asyncio applications rarely need to be written this way; consider using the high-level functions like `asyncio.run`.

Hello World with call_soon()

An example using the `loop.call_soon` method to schedule a callback. The callback displays `"Hello World"` and then stops the event loop:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# Schedule a call to hello_world()
```

```
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

A similar `Hello World <coroutine>` example created with a coroutine and the `run` function.

Display the current date with call_later()

An example of a callback displaying the current date every second. The callback uses the `loop.call_later` method to reschedule itself after 5 seconds, and then stops the event loop:

```
import asyncio
import datetime as dt

def display_date(end_time, loop):
    print(dt.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

A similar `current date <asyncio_example_sleep>` example created with a coroutine and the `run` function.

Watch a file descriptor for read events

Wait until a file descriptor received some data using the `loop.add_reader` method and then close the event loop:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

- A similar `example <asyncio_example_create_connection>` using transports, protocols, and the `loop.create_connection` method.

- Another similar `example <asyncio_example_create_connection-streams>` using the high-level `asyncio.open_connection` function and streams.

Set signal handlers for SIGINT and SIGTERM

(This `signal` example only works on Unix.)

Register handlers for signals `~signal.SIGINT` and `~signal.SIGTERM` using the `loop.add_signal_handler` method:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

□□□ #2
□ □□□ □□□ 2026-10-31 17:02:46 UTC
□ □□□ □□□ 2026-10-31 17:02:50 UTC