Event Loop

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:

`asyncio.`**`get_running_loop`**()
Return the running event loop in the current OS thread.

If there is no running event loop a `RuntimeError` is raised. This function can only be called from a coroutine or a callback.

New in version 3.7.

`asyncio.`**`get_event_loop`**()
Get the current event loop. If there is no current event loop set in the current OS thread and `set_event_loop()` has not yet been called, asyncio will create a new event loop and set it as the current one.

Because this function has rather complex behavior (especially when custom event loop policies are in use), using the `get_running_loop()` function is preferred to `get_event_loop()` in coroutines and callbacks.

Consider also using the `asyncio.run()` function instead of using lower level functions to manually create and close an event loop.

`asyncio.`**`set_event_loop`**(*loop*)
Set *loop* as a current event loop for the current OS thread.

`asyncio.`**`new_event_loop`**()
Create a new event loop object.

Note that the behaviour of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()` functions can be altered by setting a custom event loop policy.

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 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.

coroutine `loop.`**`shutdown_asyncgens`**()
Schedule all currently open asynchronous generator objects to close with an `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()
```

New in version 3.6.

Scheduling callbacks

`loop.`**`call_soon`**(*callback*, **args*, *context=None*)
Schedule a *callback* to be called with *args* arguments at the next iteration of the event loop.

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

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.

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

This method is not thread-safe.

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

A thread-safe variant of `call_soon()`. Must be used to schedule callbacks *from another thread*.

See the concurrency and multithreading section of the documentation.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 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. If you

want the callback to be called with keyword arguments use `functools.partial()`.

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.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

Changed in version 3.7.1: In Python 3.7.0 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.7.1.

`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.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** for more details.

Changed in version 3.7.1: In Python 3.7.0 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 3.7.1.

`loop.`**`time`**()

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

Note:

Changed in version 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.

See also: 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).

New in version 3.5.2.

`loop.`**`create_task`**(*coro*)

Schedule the execution of a Coroutines. 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`.

`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)`, where *loop* is a reference to the active event loop, and *coro* is a coroutine object. The callable must return a `asyncio.Future`-compatible object.

`loop.`**`get_task_factory`**()

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

Opening network connections

coroutine `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*)

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

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

The socket type will be `SOCK_STREAM`.

protocol_factory must be a callable returning an 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 is created for it.
2. *protocol_factory* is called without arguments and is expected to return a protocol instance.
3. The protocol instance is coupled with the transport by calling its `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.

  See also: SSL/TLS security considerations

- *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.
- *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* and *local_addr* should be specified.

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to 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).

New in version 3.7: The *ssl_handshake_timeout* parameter.

Changed in version 3.6: The socket option `TCP_NODELAY` is set by default for all TCP connections.

Changed in version 3.5: Added support for SSL/TLS in `ProactorEventLoop`.

See also: 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.

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

Create a datagram connection.

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

The socket type will be `SOCK_DGRAM`.

protocol_factory must be a callable returning a 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 to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`.
- *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_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 and some Unixes. If the `SO_REUSEPORT` 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`).

On Windows, with `ProactorEventLoop`, this method is not supported.

See UDP echo client protocol and UDP echo server protocol examples.

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

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

Create a Unix connection.

The socket family will be `AF_UNIX`; socket type will be `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 `Path` paths are supported.

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

Availability: Unix.

New in version 3.7: The *ssl_handshake_timeout* parameter.

Changed in version 3.7: The *path* parameter can now be a path-like object.

Creating network servers

coroutine `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*, *ssl_handshake_timeout=None*, *start_serving=True*)

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

Returns a `Server` object.

Arguments:

- *protocol_factory* must be a callable returning a 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).
- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set, the *family* will be determined from host name (defaults to `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.
- *backlog* is the maximum number of queued connections passed to `listen()` (defaults to 100).
- *ssl* can be set to an `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.
- *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).
- *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.

New in version 3.7: Added *ssl_handshake_timeout* and *start_serving* parameters.

Changed in version 3.6: The socket option `TCP_NODELAY` is set by default for all TCP connections.

Changed in version 3.5: Added support for SSL/TLS in `ProactorEventLoop`.

Changed in version 3.5.1: The *host* parameter can be a sequence of strings.

See also: 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.

coroutine `loop.`**`create_unix_server`**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Similar to `loop.create_server()` but works with the `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 `Path` paths are supported.

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

Availability: Unix.

New in version 3.7: The *ssl_handshake_timeout* and *start_serving* parameters.

Changed in version 3.7: The *path* parameter can now be a `Path` object.

coroutine `loop.`**`connect_accepted_socket`**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_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 implementation.
- *sock* is a preexisting socket object returned from `socket.accept`.
- *ssl* can be set to an `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).

Returns a `(transport, protocol)` pair.

New in version 3.7: The *ssl_handshake_timeout* parameter.

New in version 3.5.3.

Transferring files

coroutine `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()` 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`.

New in version 3.7.

TLS Upgrade

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

Upgrade an existing transport-based connection to TLS.

Return a new transport instance, that the *protocol* must start using immediately after the *await*. The *transport* instance passed to the *start_tls* method should never be used again.

Parameters:

- *transport* and *protocol* instances that methods like `create_server()` and `create_connection()` return.
- *sslcontext*: a configured instance of `SSLContext`.
- *server_side* pass `True` when a server-side connection is being upgraded (like the one created by `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).

New in version 3.7.

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.

`loop.`**`remove_reader`**(*fd*)

Stop monitoring the *fd* file descriptor for read availability.

`loop.`**`add_writer`**(*fd*, *callback*, **args*)

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

Use `functools.partial()` to pass keyword arguments to *callback*.

`loop.`**`remove_writer`**(*fd*)

Stop monitoring the *fd* file descriptor for write availability.

See also 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` objects directly is more convenient.

coroutine `loop.`**`sock_recv`**(*sock*, *nbytes*)

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

Return the received data as a bytes object.

sock must be a non-blocking socket.

Changed in version 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.

coroutine `loop.`**`sock_recv_into`**(*sock*, *buf*)

Receive data from *sock* into the *buf* buffer. Modeled after the blocking

`socket.recv_into()` method.

Return the number of bytes written to the buffer.

sock must be a non-blocking socket.

New in version 3.7.

coroutine `loop.`**`sock_sendall`**(*sock*, *data*)

Send *data* to the *sock* socket. Asynchronous version of `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.

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

coroutine `loop.`**`sock_connect`**(*sock*, *address*)

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

Asynchronous version of `socket.connect()`.

sock must be a non-blocking socket.

Changed in version 3.5.2: `address` no longer needs to be resolved. `sock_connect` will try to check if the *address* is already resolved by calling `socket.inet_pton()`. If not, `loop.getaddrinfo()` will be used to resolve the *address*.

See also: `loop.create_connection()` and `asyncio.open_connection()`.

coroutine `loop.`**`sock_accept`**(*sock*)

Accept a connection. Modeled after the blocking `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.

Changed in version 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.

See also: `loop.create_server()` and `start_server()`.

coroutine `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()`.

sock must be a non-blocking `socket.SOCK_STREAM 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()` 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.

New in version 3.7.

DNS

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

Asynchronous version of `socket.getaddrinfo()`.

coroutine `loop.`**`getnameinfo`**(*sockaddr*, *flags=0*)

Asynchronous version of `socket.getnameinfo()`.

Changed in version 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

coroutine `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 implementation.

pipe is a file-like object.

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.

coroutine `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 implementation.

pipe is file-like object.

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.

Note: `SelectorEventLoop` does not support the above methods on Windows. Use

`ProactorEventLoop` instead for Windows.

See also: 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.

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 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.

Availability: Unix.

See also: The `signal` module.

Executing code in thread or process pools

awaitable `loop.`**`run_in_executor`**(*executor*, *func*, **args*)

Arrange for *func* to be called in the specified executor.

The *executor* argument should be an `concurrent.futures.Executor` instance. The default executor is used if *executor* is `None`.

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)

asyncio.run(main())
```

This method returns a `asyncio.Future` object.

Use `functools.partial()` to pass keyword arguments to *func*.

Changed in version 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 (`ThreadPoolExecutor`) to set the default.

`loop.`**`set_default_executor`**(*executor*)
Set *executor* as the default executor used by `run_in_executor()`. *executor* should be an instance of `ThreadPoolExecutor`.

Deprecated since version 3.7: Using an executor that is not an instance of `ThreadPoolExecutor` is deprecated and will trigger an error in Python 3.9.

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).

`loop.`**`get_exception_handler`**()
Return the current exception handler, or `None` if no custom exception handler was set.

New in version 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;
- 'handle' (optional): `asyncio.Handle` instance;
- 'protocol' (optional): Protocol instance;
- 'transport' (optional): Transport instance;
- 'socket' (optional): `socket.socket` instance.

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.

Changed in version 3.7: The new `-X dev` command line option can now also be used to enable the debug mode.

See also: The debug mode of asyncio.

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: The default asyncio event loop on **Windows** does not support subprocesses. See Subprocess Support on Windows for details.

coroutine `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.

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 `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*: either a file-like object representing a pipe to be connected to the subprocess's standard input stream using `connect_write_pipe()`, or the `subprocess.PIPE` constant (default). By default a new pipe will be created and connected.
- *stdout*: either a file-like object representing the pipe to be connected to the subprocess's standard output stream using `connect_read_pipe()`, or the `subprocess.PIPE` constant (default). By default a new pipe will be created and connected.
- *stderr*: either a file-like object representing the pipe to be connected to the subprocess's standard error stream using `connect_read_pipe()`, or one of `subprocess.PIPE` (default) or `subprocess.STDOUT` constants.

By default a new pipe will be created and connected. When `subprocess.STDOUT` is specified, the subprocess' standard error stream will be connected to the same pipe as the standard output stream.
- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for *bufsize*, *universal_newlines* and *shell*, which should not be specified at all.

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*.

coroutine `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, 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 `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*.

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

class asyncio.**Handle**

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

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.

New in version 3.7.

class asyncio.**TimerHandle**

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()`.

New in version 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 class directly.

class asyncio.**Server**

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.
```

Changed in version 3.7: Server object is an asynchronous context manager since Python 3.7.

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.

get_loop()

Return the event loop associated with the server object.

New in version 3.7.

coroutine **start_serving**()

Start accepting connections.

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

New in version 3.7.

coroutine **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))
```

New in version 3.7.

is_serving()

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

New in version 3.7.

coroutine **wait_closed**()

Wait until the `close()` method completes.

sockets

List of `socket.socket` objects the server is listening on, or `None` if the server is closed.

Changed in version 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 `SelectorEventLoop` on all platforms.

class asyncio.**SelectorEventLoop**

An event loop 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

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

Availability: Unix, Windows.

class asyncio.**ProactorEventLoop**

An event loop for Windows that uses "I/O Completion Ports" (IOCP).

Availability: Windows.

An example how to use `ProactorEventLoop` on Windows:

```
import asyncio
import sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

See also: MSDN documentation on I/O Completion Ports.

class asyncio.**AbstractEventLoop**

Abstract base class for asyncio-compliant event loops.

The Event Loop Methods section lists all methods that an alternative implementation of `AbstractEventLoop` should have defined.

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 asyn-

cio 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.get_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()
```

See also: A similar Hello World 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

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

loop = asyncio.get_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()
```

See also: A similar current date 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.get_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()
```

See also:

- A similar example using transports, protocols, and the `loop.create_connection()` method.
- Another similar example using the high-level `asyncio.open_connection()` function and streams.

Set signal handlers for SIGINT and SIGTERM

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

Register handlers for signals `SIGINT` and `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())
```