This exception is a subclass of *EOFError*.

expected
The total number (*int*) of expected bytes.

partial
A string of *bytes* read before the end of stream was reached.

exception asyncio.LimitOverrunError
Reached the buffer size limit while looking for a separator.

Raised by the *asyncio stream APIs*.

consumed
The total number of to be consumed bytes.

19.1.7 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: Timeouts (relative *delay* or absolute *when*) should not exceed one day.

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 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.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):
    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))

    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())
```

19.1.8 Futures

Future objects are used to bridge **low-level callback-based code** with high-level async/await code.

Future Functions

asyncio.**isfuture**(*obj*)
Return `True` if *obj* is either of:

- an instance of *asyncio.Future*,
- an instance of *asyncio.Task*,
- a Future-like object with a `_asyncio_future_blocking` attribute.

New in version 3.5.

asyncio.**ensure_future**(*obj*, ***, *loop=None*)
Return:

- *obj* argument as is, if *obj* is a *Future*, a *Task*, or a Future-like object (*isfuture()* is used for the test.)
- a *Task* object wrapping *obj*, if *obj* is a coroutine (*iscoroutine()* is used for the test.)
- a *Task* object that would await on *obj*, if *obj* is an awaitable (*inspect.isawaitable()* is used for the test.)

If *obj* is neither of the above a *TypeError* is raised.

Important: See also the *create_task()* function which is the preferred way for creating new Tasks.

Changed in version 3.5.1: The function accepts any *awaitable* object.

asyncio.**wrap_future**(*future*, ***, *loop=None*)
Wrap a *concurrent.futures.Future* object in a *asyncio.Future* object.

Future Object

class asyncio.**Future**(***, *loop=None*)
A Future represents an eventual result of an asynchronous operation. Not thread-safe.

Future is an *awaitable* object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled.

Typically Futures are used to enable low-level callback-based code (e.g. in protocols implemented using asyncio *transports*) to interoperate with high-level async/await code.

The rule of thumb is to never expose Future objects in user-facing APIs, and the recommended way to create a Future object is to call *loop.create_future()*. This way alternative event loop implementations can inject their own optimized implementations of a Future object.

Changed in version 3.7: Added support for the *contextvars* module.

result()
Return the result of the Future.

If the Future is *done* and has a result set by the *set_result()* method, the result value is returned.

If the Future is *done* and has an exception set by the *set_exception()* method, this method raises the exception.

If the Future has been *cancelled*, this method raises a *CancelledError* exception.

If the Future's result isn't yet available, this method raises a *InvalidStateError* exception.

set_result(*result*)
Mark the Future as *done* and set its result.

Raises a *InvalidStateError* error if the Future is already *done*.

set_exception(*exception*)
Mark the Future as *done* and set an exception.

Raises a *InvalidStateError* error if the Future is already *done*.

done()
Return **True** if the Future is *done*.

A Future is *done* if it was *cancelled* or if it has a result or an exception set with *set_result()* or *set_exception()* calls.

cancelled()
Return **True** if the Future was *cancelled*.

The method is usually used to check if a Future is not *cancelled* before setting a result or an exception for it:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, ***, *context=None*)
Add a callback to be run when the Future is *done*.

The *callback* is called with the Future object as its only argument.

If the Future is already *done* when this method is called, the callback is scheduled with *loop.call_soon()*.

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.

functools.partial() can be used to pass parameters to the callback, e.g.:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

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

remove_done_callback(*callback*)
Remove *callback* from the callbacks list.

Returns the number of callbacks removed, which is typically 1, unless a callback was added more than once.

cancel()
Cancel the Future and schedule callbacks.

If the Future is already *done* or *cancelled*, return `False`. Otherwise, change the Future's state to *cancelled*, schedule the callbacks, and return `True`.

exception()
Return the exception that was set on this Future.

The exception (or `None` if no exception was set) is returned only if the Future is *done*.

If the Future has been *cancelled*, this method raises a *CancelledError* exception.

If the Future isn't *done* yet, this method raises an *InvalidStateError* exception.

get_loop()
Return the event loop the Future object is bound to.

New in version 3.7.

This example creates a Future object, creates and schedules an asynchronous Task to set result for the Future, and waits until the Future has a result:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
```

(continues on next page)

(continued from previous page)

```
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

Important: The Future object was designed to mimic *concurrent.futures.Future*. Key differences include:

- unlike asyncio Futures, *concurrent.futures.Future* instances cannot be awaited.
- *asyncio.Future.result()* and *asyncio.Future.exception()* do not accept the *timeout* argument.
- *asyncio.Future.result()* and *asyncio.Future.exception()* raise an *InvalidStateError* exception when the Future is not *done*.
- Callbacks registered with *asyncio.Future.add_done_callback()* are not called immediately. They are scheduled with *loop.call_soon()* instead.
- asyncio Future is not compatible with the *concurrent.futures.wait()* and *concurrent.futures.as_completed()* functions.

19.1.9 Transports and Protocols

Preface

Transports and Protocols are used by the **low-level** event loop APIs such as *loop.create_connection()*. They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both *Transports* and *Protocols*.

Introduction

At the highest level, the transport is concerned with *how* bytes are transmitted, while the protocol determines *which* bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport's point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

Most of connection oriented event loop methods (such as *loop.create_connection()*) usually accept a *protocol_factory* argument used to create a *Protocol* object for an accepted connection, represented by a *Transport* object. Such methods usually return a tuple of `(transport, protocol)`.

Contents

This documentation page contains the following sections:

- The *Transports* section documents asyncio *BaseTransport*, *ReadTransport*, *WriteTransport*, *Transport*, *DatagramTransport*, and *SubprocessTransport* classes.
- The *Protocols* section documents asyncio *BaseProtocol*, *Protocol*, *BufferedProtocol*, *DatagramProtocol*, and *SubprocessProtocol* classes.
- The *Examples* section showcases how to work with transports, protocols, and low-level event loop APIs.

Transports

Transports are classes provided by *asyncio* in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an ref:*asyncio event loop <asyncio-event-loop>*.

asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are *not thread safe*.

Transports Hierarchy

class asyncio.BaseTransport
Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.WriteTransport(*BaseTransport*)
A base transport for write-only connections.

Instances of the *WriteTransport* class are returned from the *loop.connect_write_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.ReadTransport(*BaseTransport*)
A base transport for read-only connections.

Instances of the *ReadTransport* class are returned from the *loop.connect_read_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.Transport(*WriteTransport*, *ReadTransport*)
Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the *Transport* class are returned from or used by event loop methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.create_server()*, *loop.sendfile()*, etc.

class asyncio.DatagramTransport(*BaseTransport*)
A transport for datagram (UDP) connections.

Instances of the *DatagramTransport* class are returned from the *loop.create_datagram_endpoint()* event loop method.

class asyncio.SubprocessTransport(*BaseTransport*)
An abstraction to represent a connection between a parent and its child OS process.

Instances of the *SubprocessTransport* class are returned from event loop methods *loop.subprocess_shell()* and *loop.subprocess_exec()*.

Base Transport

BaseTransport.close()
Close the transport.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *protocol.connection_lost()* method will be called with *None* as its argument.

BaseTransport.is_closing()
Return True if the transport is closing or is closed.

BaseTransport.get_extra_info(*name*, *default=None*)
Return information about the transport or underlying resources it uses.

name is a string representing the piece of transport-specific information to get.

default is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
  - 'peername': the remote address to which the socket is connected, result of *socket.socket.getpeername()* (None on error)
  - 'socket': *socket.socket* instance
  - 'sockname': the socket's own address, result of *socket.socket.getsockname()*
- SSL socket:
  - 'compression': the compression algorithm being used as a string, or None if the connection isn't compressed; result of *ssl.SSLSocket.compression()*
  - 'cipher': a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of *ssl.SSLSocket.cipher()*
  - 'peercert': peer certificate; result of *ssl.SSLSocket.getpeercert()*
  - 'sslcontext': *ssl.SSLContext* instance
  - 'ssl_object': *ssl.SSLObject* or *ssl.SSLSocket* instance
- pipe:
  - 'pipe': pipe object
- subprocess:
  - 'subprocess': *subprocess.Popen* instance

BaseTransport.**set_protocol**(*protocol*)
Set a new protocol.

Switching protocol should only be done when both protocols are documented to support the switch.

BaseTransport.**get_protocol**()
Return the current protocol.

Read-only Transports

ReadTransport.**is_reading**()
Return `True` if the transport is receiving new data.

New in version 3.7.

ReadTransport.**pause_reading**()
Pause the receiving end of the transport. No data will be passed to the protocol's *protocol.data_received()* method until *resume_reading()* is called.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

ReadTransport.**resume_reading**()
Resume the receiving end. The protocol's *protocol.data_received()* method will be called once again if some data is available for reading.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

WriteTransport.**abort**()
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

WriteTransport.**can_write_eof**()
Return *True* if the transport supports *write_eof()*, *False* if not.

WriteTransport.**get_write_buffer_size**()
Return the current size of the output buffer used by the transport.

WriteTransport.**get_write_buffer_limits**()
Get the *high* and *low* watermarks for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use *set_write_buffer_limits()* to set the limits.

New in version 3.4.2.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)
Set the *high* and *low* watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's *protocol.pause_writing()* and *protocol.resume_writing()* methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither *high* nor *low* can be negative.

pause_writing() is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, *resume_writing()* is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high watermark. Setting *high* to zero forces *low* to zero as well, and causes *pause_writing()* to be called whenever the buffer becomes non-empty. Setting *low* to zero causes *resume_writing()* to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

`WriteTransport.write(`*data*`)`
Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

`WriteTransport.writelines(`*list_of_data*`)`
Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling *write()* on each element yielded by the iterable, but may be implemented more efficiently.

`WriteTransport.write_eof()`
Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closed connections.

Datagram Transports

`DatagramTransport.sendto(`*data*, *addr=None*`)`
Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is *None*, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

`DatagramTransport.abort()`
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

Subprocess Transports

`SubprocessTransport.get_pid()`
Return the subprocess process id as an integer.

`SubprocessTransport.get_pipe_transport(`*fd*`)`
Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- `0`: readable streaming transport of the standard input (*stdin*), or *None* if the subprocess was not created with `stdin=PIPE`
- `1`: writable streaming transport of the standard output (*stdout*), or *None* if the subprocess was not created with `stdout=PIPE`
- `2`: writable streaming transport of the standard error (*stderr*), or *None* if the subprocess was not created with `stderr=PIPE`
- other *fd*: *None*

`SubprocessTransport.get_returncode()`
Return the subprocess return code as an integer or *None* if it hasn't returned, which is similar to the *subprocess.Popen.returncode* attribute.

`SubprocessTransport.kill()`
Kill the subprocess.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for *terminate()*.

See also *subprocess.Popen.kill()*.

`SubprocessTransport.send_signal(`*signal*`)`
Send the *signal* number to the subprocess, as in *subprocess.Popen.send_signal()*.

`SubprocessTransport.terminate()`
Stop the subprocess.

On POSIX systems, this method sends SIGTERM to the subprocess. On Windows, the Windows API function TerminateProcess() is called to stop the subprocess.

See also *subprocess.Popen.terminate()*.

`SubprocessTransport.close()`
Kill the subprocess by calling the *kill()* method.

If the subprocess hasn't returned yet, and close transports of *stdin*, *stdout*, and *stderr* pipes.

Protocols

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with *transports*.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Base Protocols

`class asyncio.BaseProtocol`
Base protocol with methods that all protocols share.

`class asyncio.Protocol(`*BaseProtocol*`)`
The base class for implementing streaming protocols (TCP, Unix sockets, etc).

`class asyncio.BufferedProtocol(`*BaseProtocol*`)`
A base class for implementing streaming protocols with manual control of the receive buffer.

`class asyncio.DatagramProtocol(`*BaseProtocol*`)`
The base class for implementing datagram (UDP) protocols.

`class asyncio.SubprocessProtocol(`*BaseProtocol*`)`
The base class for implementing protocols communicating with child processes (unidirectional pipes).

Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

Connection Callbacks

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

BaseProtocol.**connection_made**(*transport*)
Called when a connection is made.

The *transport* argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

BaseProtocol.**connection_lost**(*exc*)
Called when the connection is lost or closed.

The argument is either an exception object or *None*. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

Flow Control Callbacks

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the *set_write_buffer_limits()* method for more details.

BaseProtocol.**pause_writing**()
Called when the transport's buffer goes over the high watermark.

BaseProtocol.**resume_writing**()
Called when the transport's buffer drains below the low watermark.

If the buffer size equals the high watermark, *pause_writing()* is not called: the buffer size must go strictly over.

Conversely, *resume_writing()* is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

Streaming Protocols

Event methods, such as *loop.create_server()*, *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()*, and *loop.connect_write_pipe()* accept factories that return streaming protocols.

Protocol.**data_received**(*data*)
Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.

The method can be called an arbitrary number of times while a connection is open.

However, *protocol.eof_received()* is called at most once. Once *eof_received()* is called, `data_received()` is not called anymore.

Protocol.**eof_received**()
Called when the other end signals it won't send any more data (for example by calling *transport.write_eof()*, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don't support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Buffered Streaming Protocols

New in version 3.7: **Important:** this has been added to asyncio in Python 3.7 *on a provisional basis*! This is as an experimental API that might be changed or removed completely in Python 3.8.

Buffered Protocols can be used with any event loop method that supports *Streaming Protocols*.

`BufferedProtocol` implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on *`BufferedProtocol`* instances:

`BufferedProtocol.`**`get_buffer`**`(`*`sizehint`*`)`
Called to allocate a new receive buffer.

sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what *sizehint* suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

`get_buffer()` must return an object implementing the buffer protocol.

`BufferedProtocol.`**`buffer_updated`**`(`*`nbytes`*`)`
Called when the buffer was updated with the received data.

nbytes is the total number of bytes that were written to the buffer.

`BufferedProtocol.`**`eof_received`**`()`
See the documentation of the *`protocol.eof_received()`* method.

`get_buffer()` can be called an arbitrary number of times during a connection. However, *`protocol.eof_received()`* is called at most once and, if called, *`get_buffer()`* and *`buffer_updated()`* won't be called after it.

State machine:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Datagram Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the *`loop.create_datagram_endpoint()`* method.

`DatagramProtocol.`**`datagram_received`**`(`*`data`*`, `*`addr`*`)`
Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

DatagramProtocol.**error_received**(*exc*)
Called when a previous send or receive operation raises an *OSError*. *exc* is the *OSError* instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Note: On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears 'ready' and excess packets are dropped. An *OSError* with `errno` set to *errno.ENOBUFS* may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Subprocess Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the *loop.subprocess_exec()* and *loop.subprocess_shell()* methods.

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)
Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)
Called when one of the pipes communicating with the child process is closed.

fd is the integer file descriptor that was closed.

SubprocessProtocol.**process_exited**()
Called when the child process has exited.

Examples

TCP Echo Server

Create a TCP echo server using the *loop.create_server()* method, send back received data, and close the connection:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
```

(continues on next page)

(continued from previous page)

```
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

See also:

The *TCP echo server using streams* example uses the high-level *asyncio.start_server()* function.

TCP Echo Client

A TCP echo client using the *loop.create_connection()* method, sends data, and waits until the connection is closed:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost, loop):
        self.message = message
        self.loop = loop
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
```

(continues on next page)

(continued from previous page)

```
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost, loop),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

See also:

The *TCP echo client using streams* example uses the high-level *asyncio.open_connection()* function.

UDP Echo Server

A UDP echo server, using the *loop.create_datagram_endpoint()* method, sends back received data:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
```

(continues on next page)

(continued from previous page)

```
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

UDP Echo Client

A UDP echo client, using the *loop.create_datagram_endpoint()* method, sends data and closes the transport when it receives the answer:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None
        self.on_con_lost = loop.create_future()

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
```

(continues on next page)

(continued from previous page)

```
    message = "Hello World!"
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, loop),
        remote_addr=('127.0.0.1', 9999))

    try:
        await protocol.on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the *loop.create_connection()* method with a protocol:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, loop):
        self.transport = None
        self.on_con_lost = loop.create_future()

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
```

(continues on next page)

(continued from previous page)

```
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(loop), sock=rsock)

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

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

See also:

The *watch a file descriptor for read events* example uses the low-level *loop.add_reader()* method to register an FD.

The *register an open socket to wait for data using streams* example uses high-level streams created by the *open_connection()* function in a coroutine.

loop.subprocess_exec() and SubprocessProtocol

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by th *loop.subprocess_exec()* method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
```

(continues on next page)

(continued from previous page)

```
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using high-level APIs.

19.1.10 Policies

An event loop policy is a global per-process object that controls the management of the event loop. Each event loop has a default policy, which can be changed and customized using the policy API.

A policy defines the notion of *context* and manages a separate event loop per context. The default policy defines *context* to be the current thread.

By using a custom event loop policy, the behavior of *get_event_loop()*, *set_event_loop()*, and *new_event_loop()* functions can be customized.

Policy objects should implement the APIs defined in the *AbstractEventLoopPolicy* abstract base class.

Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

asyncio.get_event_loop_policy()
Return the current process-wide policy.

asyncio.set_event_loop_policy(*policy*)
Set the current process-wide policy to *policy*.

If *policy* is set to `None`, the default policy is restored.

Policy Objects

The abstract event loop policy base class is defined as follows:

class asyncio.AbstractEventLoopPolicy
An abstract base class for asyncio policies.

get_event_loop()
Get the event loop for the current context.

Return an event loop object implementing the *AbstractEventLoop* interface.

This method should never return `None`.

Changed in version 3.6.

set_event_loop(*loop*)
Set the event loop for the current context to *loop*.

new_event_loop()
Create and return a new event loop object.

This method should never return `None`.

get_child_watcher()
Get a child process watcher object.

Return a watcher object implementing the *AbstractChildWatcher* interface.

This function is Unix specific.

set_child_watcher(*watcher*)
Set the current child process watcher to *watcher*.

This function is Unix specific.

asyncio ships with the following built-in policies:

class asyncio.DefaultEventLoopPolicy
The default asyncio policy. Uses *SelectorEventLoop* on both Unix and Windows platforms.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

class asyncio.WindowsProactorEventLoopPolicy
An alternative event loop policy that uses the *ProactorEventLoop* event loop implementation.

Availability: Windows.

Process Watchers

A process watcher allows customization of how an event loop monitors child processes on Unix. Specifically, the event loop needs to know when a child process has exited.

In asyncio, child processes are created with *create_subprocess_exec()* and *loop.subprocess_exec()* functions.

asyncio defines the *AbstractChildWatcher* abstract base class, which child watchers should implement, and has two different implementations: *SafeChildWatcher* (configured to be used by default) and *FastChildWatcher*.

See also the *Subprocess and Threads* section.

The following two functions can be used to customize the child process watcher implementation used by the asyncio event loop:

asyncio.get_child_watcher()
Return the current child watcher for the current policy.

asyncio.**set_child_watcher**(*watcher*)
Set the current child watcher to *watcher* for the current policy. *watcher* must implement methods defined in the *AbstractChildWatcher* base class.

Note: Third-party event loops implementations might not support custom child watchers. For such event loops, using *set_child_watcher()* might be prohibited or have no effect.

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)
Register a new child handler.

Arrange for `callback(pid, returncode, *args)` to be called when a process with PID equal to *pid* terminates. Specifying another callback for the same process replaces the previous handler.

The *callback* callable must be thread-safe.

remove_child_handler(*pid*)
Removes the handler for process with PID equal to *pid*.

The function returns `True` if the handler was successfully removed, `False` if there was nothing to remove.

attach_loop(*loop*)
Attach the watcher to an event loop.

If the watcher was previously attached to an event loop, then it is first detached before attaching to the new loop.

Note: loop may be `None`.

close()
Close the watcher.

This method has to be called to ensure that underlying resources are cleaned-up.

class asyncio.**SafeChildWatcher**
This implementation avoids disrupting other code spawning processes by polling every process explicitly on a `SIGCHLD` signal.

This is a safe solution but it has a significant overhead when handling a big number of processes (*O(n)* each time a `SIGCHLD` is received).

asyncio uses this safe implementation by default.

class asyncio.**FastChildWatcher**
This implementation reaps every terminated processes by calling `os.waitpid(-1)` directly, possibly breaking other code spawning processes and waiting for their termination.

There is no noticeable overhead when handling a big number of children (*O(1)* each time a child terminates).

Custom Policies

To implement a new event loop policy, it is recommended to subclass *DefaultEventLoopPolicy* and override the methods for which custom behavior is wanted, e.g.:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
```

(continues on next page)

(continued from previous page)

```
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.11 Platform Support

The *asyncio* module is designed to be portable, but some platforms have subtle differences and limitations due to the platforms' underlying architecture and capabilities.

All Platforms

- *loop.add_reader()* and *loop.add_writer()* cannot be used to monitor file I/O.

Windows

All event loops on Windows do not support the following methods:

- *loop.create_unix_connection()* and *loop.create_unix_server()* are not supported. The *socket.AF_UNIX* socket family is specific to Unix.
- *loop.add_signal_handler()* and *loop.remove_signal_handler()* are not supported.

SelectorEventLoop has the following limitations:

- *SelectSelector* is used to wait on socket events: it supports sockets and is limited to 512 sockets.
- *loop.add_reader()* and *loop.add_writer()* only accept socket handles (e.g. pipe file descriptors are not supported).
- Pipes are not supported, so the *loop.connect_read_pipe()* and *loop.connect_write_pipe()* methods are not implemented.
- *Subprocesses* are not supported, i.e. *loop.subprocess_exec()* and *loop.subprocess_shell()* methods are not implemented.

ProactorEventLoop has the following limitations:

- The *loop.create_datagram_endpoint()* method is not supported.
- The *loop.add_reader()* and *loop.add_writer()* methods are not supported.

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows configuration.

Subprocess Support on Windows

SelectorEventLoop on Windows does not support subproceses. On Windows, *ProactorEventLoop* should be used instead:

```
import asyncio

asyncio.set_event_loop_policy(
    asyncio.WindowsProactorEventLoopPolicy())

asyncio.run(your_code())
```

The *policy.set_child_watcher()* function is also not supported, as *ProactorEventLoop* has a different mechanism to watch child processes.

macOS

Modern macOS versions are fully supported.

macOS <= 10.8

On macOS 10.6, 10.7 and 10.8, the default event loop uses *selectors.KqueueSelector*, which does not support character devices on these versions. The *SelectorEventLoop* can be manually configured to use *SelectSelector* or *PollSelector* to support character devices on these older versions of macOS. Example:

```
import asyncio
import selectors

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

19.1.12 High-level API Index

This page lists all high-level async/await enabled asyncio APIs.

Tasks

Utilities to run asyncio programs, create Tasks, and await on multiple things with timeouts.

run()	Create event loop, run a coroutine, close the loop.
create_task()	Start an asyncio Task.
await *sleep()*	Sleep for a number of seconds.
await *gather()*	Schedule and wait for things concurrently.
await *wait_for()*	Run with a timeout.
await *shield()*	Shield from cancellation.
await *wait()*	Monitor for completion.
current_task()	Return the current Task.
all_tasks()	Return all tasks for an event loop.
Task	Task object.
run_coroutine_threadsafe()	Schedule a coroutine from another OS thread.
for in *as_completed()*	Monitor for completion with a `for` loop.

Examples

- *Using asyncio.gather() to run things in parallel*.
- *Using asyncio.wait_for() to enforce a timeout*.
- *Cancellation*.
- *Using asyncio.sleep()*.
- See also the main *Tasks documentation page*.

Queues

Queues should be used to distribute work amongst multiple asyncio Tasks, implement connection pools, and pub/sub patterns.

Queue	A FIFO queue.
PriorityQueue	A priority queue.
LifoQueue	A LIFO queue.

Examples

- *Using asyncio.Queue to distribute workload between several Tasks*.
- See also the *Queues documentation page*.

Subprocesses

Utilities to spawn subprocesses and run shell commands.

`await` *create_subprocess_exec()*	Create a subprocess.
`await` *create_subprocess_shell()*	Run a shell command.

Examples

- *Executing a shell command*.
- See also the *subprocess APIs* documentation.

Streams

High-level APIs to work with network IO.

`await` *open_connection()*	Establish a TCP connection.
`await` *open_unix_connection()*	Establish a Unix socket connection.
`await` *start_server()*	Start a TCP server.
`await` *start_unix_server()*	Start a Unix socket server.
StreamReader	High-level async/await object to receive network data.
StreamWriter	High-level async/await object to send network data.

Examples

- *Example TCP client*.
- See also the *streams APIs* documentation.

Synchronization

Threading-like synchronization primitives that can be used in Tasks.

Lock	A mutex lock.
Event	An event object.
Condition	A condition object.
Semaphore	A semaphore.
BoundedSemaphore	A bounded semaphore.

Examples

- *Using asyncio.Event*.
- See also the documentation of asyncio *synchronization primitives*.

Exceptions

asyncio.TimeoutError	Raised on timeout by functions like *wait_for()*. Keep in mind that `asyncio.TimeoutError` is **unrelated** to the built-in *TimeoutError* exception.
asyncio.CancelledError	Raised when a Task is cancelled. See also *Task.cancel()*.

Examples

- *Handling CancelledError to run code on cancellation request*.
- See also the full list of *asyncio-specific exceptions*.

19.1.13 Low-level API Index

This page lists all low-level asyncio APIs.

Obtaining the Event Loop

asyncio.get_running_loop()	The **preferred** function to get the running event loop.
asyncio.get_event_loop()	Get an event loop instance (current or via the policy).
asyncio.set_event_loop()	Set the event loop as current via the current policy.
asyncio.new_event_loop()	Create a new event loop.

Examples

- *Using asyncio.get_running_loop()*.

Event Loop Methods

See also the main documentation section about the *event loop methods*.

Lifecycle

loop.run_until_complete()	Run a Future/Task/awaitable until complete.
loop.run_forever()	Run the event loop forever.
loop.stop()	Stop the event loop.
loop.close()	Close the event loop.
loop.is_running()	Return `True` if the event loop is running.
loop.is_closed()	Return `True` if the event loop is closed.
`await` *loop.shutdown_asyncgens()*	Close asynchronous generators.

Debugging

loop.set_debug()	Enable or disable the debug mode.
loop.get_debug()	Get the current debug mode.

Scheduling Callbacks

loop.call_soon()	Invoke a callback soon.
loop.call_soon_threadsafe()	A thread-safe variant of *loop.call_soon()*.
loop.call_later()	Invoke a callback *after* the given time.
loop.call_at()	Invoke a callback *at* the given time.

Thread/Process Pool

`await` *loop.run_in_executor()*	Run a CPU-bound or other blocking function in a *concurrent.futures* executor.
loop.set_default_executor()	Set the default executor for *loop.run_in_executor()*.

Tasks and Futures

loop.create_future()	Create a *Future* object.
loop.create_task()	Schedule coroutine as a *Task*.
loop.set_task_factory()	Set a factory used by *loop.create_task()* to create *Tasks*.
loop.get_task_factory()	Get the factory *loop.create_task()* uses to create *Tasks*.

DNS

await *loop.getaddrinfo()*	Asynchronous version of *socket.getaddrinfo()*.
await *loop.getnameinfo()*	Asynchronous version of *socket.getnameinfo()*.

Networking and IPC

await *loop.create_connection()*	Open a TCP connection.
await *loop.create_server()*	Create a TCP server.
await *loop.create_unix_connection()*	Open a Unix socket connection.
await *loop.create_unix_server()*	Create a Unix socket server.
await *loop.connect_accepted_socket()*	Wrap a *socket* into a `(transport, protocol)` pair.
await *loop.create_datagram_endpoint()*	Open a datagram (UDP) connection.
await *loop.sendfile()*	Send a file over a transport.
await *loop.start_tls()*	Upgrade an existing connection to TLS.
await *loop.connect_read_pipe()*	Wrap a read end of a pipe into a `(transport, protocol)` pair.
await *loop.connect_write_pipe()*	Wrap a write end of a pipe into a `(transport, protocol)` pair.

Sockets

await *loop.sock_recv()*	Receive data from the *socket*.
await *loop.sock_recv_into()*	Receive data from the *socket* into a buffer.
await *loop.sock_sendall()*	Send data to the *socket*.
await *loop.sock_connect()*	Connect the *socket*.
await *loop.sock_accept()*	Accept a *socket* connection.
await *loop.sock_sendfile()*	Send a file over the *socket*.
loop.add_reader()	Start watching a file descriptor for read availability.
loop.remove_reader()	Stop watching a file descriptor for read availability.
loop.add_writer()	Start watching a file descriptor for write availability.
loop.remove_writer()	Stop watching a file descriptor for write availability.

Unix Signals

loop.add_signal_handler()	Add a handler for a *signal*.
loop.remove_signal_handler()	Remove a handler for a *signal*.

Subprocesses

loop.subprocess_exec()	Spawn a subprocess.
loop.subprocess_shell()	Spawn a subprocess from a shell command.

Error Handling

loop.call_exception_handler()	Call the exception handler.
loop.set_exception_handler()	Set a new exception handler.
loop.get_exception_handler()	Get the current exception handler.
loop.default_exception_handler()	The default exception handler implementation.

Examples

- *Using asyncio.get_event_loop() and loop.run_forever()*.
- *Using loop.call_later()*.
- Using `loop.create_connection()` to implement *an echo-client*.
- Using `loop.create_connection()` to *connect a socket*.
- *Using add_reader() to watch an FD for read events*.
- *Using loop.add_signal_handler()*.
- *Using loop.subprocess_exec()*.

Transports

All transports implement the following methods:

transport.close()	Close the transport.
transport.is_closing()	Return `True` if the transport is closing or is closed.
transport.get_extra_info()	Request for information about the transport.
transport.set_protocol()	Set a new protocol.
transport.get_protocol()	Return the current protocol.

Transports that can receive data (TCP and Unix connections, pipes, etc). Returned from methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_read_pipe()*, etc:

Read Transports

transport.is_reading()	Return `True` if the transport is receiving.
transport.pause_reading()	Pause receiving.
transport.resume_reading()	Resume receiving.

Transports that can Send data (TCP and Unix connections, pipes, etc). Returned from methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_write_pipe()*, etc:

Write Transports

transport.write()	Write data to the transport.
transport.writelines()	Write buffers to the transport.
transport.can_write_eof()	Return *True* if the transport supports sending EOF.
transport.write_eof()	Close and send EOF after flushing buffered data.
transport.abort()	Close the transport immediately.
transport.get_write_buffer_size()	Return high and low water marks for write flow control.
transport.set_write_buffer_limits()	Set new high and low water marks for write flow control.

Transports returned by *loop.create_datagram_endpoint()*:

Datagram Transports

transport.sendto()	Send data to the remote peer.
transport.abort()	Close the transport immediately.

Low-level transport abstraction over subprocesses. Returned by *loop.subprocess_exec()* and *loop.subprocess_shell()*:

Subprocess Transports

transport.get_pid()	Return the subprocess process id.
transport.get_pipe_transport()	Return the transport for the requested communication pipe (*stdin*, *stdout*, or *stderr*).
transport.get_returncode()	Return the subprocess return code.
transport.kill()	Kill the subprocess.
transport.send_signal()	Send a signal to the subprocess.
transport.terminate()	Stop the subprocess.
transport.close()	Kill the subprocess and close all pipes.

Protocols

Protocol classes can implement the following **callback methods**:

callback *connection_made()*	Called when a connection is made.
callback *connection_lost()*	Called when the connection is lost or closed.
callback *pause_writing()*	Called when the transport's buffer goes over the high water mark.
callback *resume_writing()*	Called when the transport's buffer drains below the low water mark.

Streaming Protocols (TCP, Unix Sockets, Pipes)

callback *data_received()*	Called when some data is received.
callback *eof_received()*	Called when an EOF is received.

Buffered Streaming Protocols

callback *get_buffer()*	Called to allocate a new receive buffer.
callback *buffer_updated()*	Called when the buffer was updated with the received data.
callback *eof_received()*	Called when an EOF is received.

Datagram Protocols

callback *datagram_received()*	Called when a datagram is received.
callback *error_received()*	Called when a previous send or receive operation raises an *OSError*.

Subprocess Protocols

callback *pipe_data_received()*	Called when the child process writes data into its *stdout* or *stderr* pipe.
callback *pipe_connection_lost()*	Called when one of the pipes communicating with the child process is closed.
callback *process_exited()*	Called when the child process has exited.

Event Loop Policies

Policies is a low-level mechanism to alter the behavior of functions like *asyncio.get_event_loop()*. See also the main *policies section* for more details.

Accessing Policies

asyncio.get_event_loop_policy()	Return the current process-wide policy.
asyncio.set_event_loop_policy()	Set a new process-wide policy.
AbstractEventLoopPolicy	Base class for policy objects.

19.1.14 Developing with asyncio

Asynchronous programming is different from classic "sequential" programming.

This page lists common mistakes and traps and explains how to avoid them.

Debug Mode

By default asyncio runs in production mode. In order to ease the development asyncio has a *debug mode*.

There are several ways to enable asyncio debug mode:

- Setting the `PYTHONASYNCIODEBUG` environment variable to `1`.
- Using the `-X dev` Python command line option.
- Passing `debug=True` to *asyncio.run()*.
- Calling *loop.set_debug()*.

In addition to enabling the debug mode, consider also:

- setting the log level of the *asyncio logger* to `logging.DEBUG`, for example the following snippet of code can be run at startup of the application:

```
logging.basicConfig(level=logging.DEBUG)
```

- configuring the *warnings* module to display *ResourceWarning* warnings. One way of doing that is by using the `-W default` command line option.

When the debug mode is enabled:

- asyncio checks for *coroutines that were not awaited* and logs them; this mitigates the "forgotten await" pitfall.
- Many non-threadsafe asyncio APIs (such as *loop.call_soon()* and *loop.call_at()* methods) raise an exception if they are called from a wrong thread.
- The execution time of the I/O selector is logged if it takes too long to perform an I/O operation.
- Callbacks taking longer than 100ms are logged. The `loop.slow_callback_duration` attribute can be used to set the minimum execution duration in seconds that is considered "slow".

Concurrency and Multithreading

An event loop runs in a thread (typically the main thread) and executes all callbacks and Tasks in its thread. While a Task is running in the event loop, no other Tasks can run in the same thread. When a Task executes an `await` expression, the running Task gets suspended, and the event loop executes the next Task.

To schedule a callback from a different OS thread, the *loop.call_soon_threadsafe()* method should be used. Example:

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

Almost all asyncio objects are not thread safe, which is typically not a problem unless there is code that works with them from outside of a Task or a callback. If there's a need for such code to call a low-level asyncio API, the *loop.call_soon_threadsafe()* method should be used, e.g.:

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

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

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:
```

(continues on next page)

(continued from previous page)

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

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

The *loop.run_in_executor()* method can be used with a *concurrent.futures.ThreadPoolExecutor* to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

Running Blocking Code

Blocking (CPU-bound) code should not be called directly. For example, if a function performs a CPU-intensive calculation for 1 second, all concurrent asyncio Tasks and IO operations would be delayed by 1 second.

An executor can be used to run a task in a different thread or even in a different process to avoid blocking block the OS thread with the event loop. See the *loop.run_in_executor()* method for more details.

Logging

asyncio uses the *logging* module and all logging is performed via the `"asyncio"` logger.

The default log level is `logging.INFO`, which can be easily adjusted:

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

Detect never-awaited coroutines

When a coroutine function is called, but not awaited (e.g. `coro()` instead of `await coro()`) or the coroutine is not scheduled with *asyncio.create_task()*, asyncio will emit a *RuntimeWarning*:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Output:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

Output in debug mode:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
```

(continues on next page)

(continued from previous page)

```
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

The usual fix is to either await the coroutine or call the *asyncio.create_task()* function:

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

Detect never-retrieved exceptions

If a *Future.set_exception()* is called but the Future object is never awaited on, the exception would never be propagated to the user code. In this case, asyncio would emit a log message when the Future object is garbage collected.

Example of an unhandled exception:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

Output:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode to get the traceback where the task was created:

```
asyncio.run(main(), debug=True)
```

Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

19.2 socket — Low-level networking interface

Source code: Lib/socket.py

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the *socket()* function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also:

Module *socketserver* Classes that simplify writing network servers.

Module *ssl* A TLS/SSL wrapper for socket objects.

19.2.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an *AF_UNIX* socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Changed in version 3.3: Previously, *AF_UNIX* socket paths were assumed to use UTF-8 encoding.

  Changed in version 3.5: Writable *bytes-like object* is now accepted.