18.5.1. Base Event Loop

Source code: Lib/asyncio/events.py

The event loop is the central execution device provided by `asyncio`. It provides multiple facilities, including:

- Registering, executing and cancelling delayed calls (timeouts).
- Creating client and server transports for various kinds of communication.
- Launching subprocesses and the associated transports for communication with an external program.
- Delegating costly function calls to a pool of threads.

class `asyncio.`**`BaseEventLoop`**

This class is an implementation detail. It is a subclass of `AbstractEventLoop` and may be a base class of concrete event loop implementations found in `asyncio`. It should not be used directly; use `AbstractEventLoop` instead. `BaseEventLoop` should not be subclassed by third-party code; the internal interface is not stable.

class `asyncio.`**`AbstractEventLoop`**

Abstract base class of event loops.

This class is not thread safe.

18.5.1.1. Run an event loop

`AbstractEventLoop.`**`run_forever`**()

Run until `stop()` is called. If `stop()` is called before `run_forever()` is called, this polls the I/O selector once with a timeout of zero, runs all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exits. If `stop()` is called while `run_forever()` is running, this will run the current batch of callbacks and then exit. Note that callbacks scheduled by callbacks will not run in that case; they will run the next time `run_forever()` is called.

Changed in version 3.5.1.

`AbstractEventLoop.`**`run_until_complete`**(*future*)

Run until the `Future` is done.

If the argument is a coroutine object, it is wrapped by `ensure_future()`.

Return the Future's result, or raise its exception.

AbstractEventLoop.**is_running**()

Returns running status of event loop.

AbstractEventLoop.**stop**()

Stop running the event loop.

This causes `run_forever()` to exit at the next suitable opportunity (see there for more details).

Changed in version 3.5.1.

AbstractEventLoop.**is_closed**()

Returns `True` if the event loop was closed.

New in version 3.4.2.

AbstractEventLoop.**close**()

Close the event loop. The loop must not be running. Pending callbacks will be lost.

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

This is idempotent and irreversible. No other methods should be called after this one.

coroutine AbstractEventLoop.**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 whenever a new asynchronous generator is iterated. Should be used to finalize all scheduled asynchronous generators reliably. Example:

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

New in version 3.6.

18.5.1.2. Calls

Most `asyncio` functions don't accept keywords. If you want to pass keywords to your callback, use `functools.partial()`. For example, `loop.call_soon(functools.partial(print, "Hello", flush=True))` will call `print("Hello", flush=True)`.

Note: `functools.partial()` is better than `lambda` functions, because `asyncio` can inspect `functools.partial()` object to display parameters in debug mode, whereas `lambda` functions have a poor representation.

`AbstractEventLoop.`**`call_soon`**(*callback*, **args*)

Arrange for a callback to be called as soon as possible. The callback is called after `call_soon()` returns, when control returns to the event loop.

This operates as a FIFO queue, callbacks are called in the order in which they are registered. Each callback will be called exactly once.

Any positional arguments after the callback will be passed to the callback when it is called.

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

Use functools.partial to pass keywords to the callback.

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

Like `call_soon()`, but thread safe.

See the concurrency and multithreading section of the documentation.

18.5.1.3. Delayed calls

The event loop has its own internal clock for computing timeouts. Which clock is used depends on the (platform-specific) event loop implementation; ideally it is a monotonic clock. This will generally be a different clock than `time.time()`.

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

`AbstractEventLoop.`**`call_later`**(*delay*, *callback*, **args*)

Arrange for the *callback* to be called after the given *delay* seconds (either an int or float).

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

callback will be called exactly once per call to `call_later()`. If two callbacks are scheduled for exactly the same time, it is undefined which will be called first.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with some named arguments, use a closure or `functools.partial()`.

Use functools.partial to pass keywords to the callback.

`AbstractEventLoop.`**`call_at`**(*when*, *callback*, **args*)

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

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

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

Use functools.partial to pass keywords to the callback.

`AbstractEventLoop.`**`time`**()

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

See also: The `asyncio.sleep()` function.

18.5.1.4. Futures

`AbstractEventLoop.`**`create_future`**()

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

This is a preferred way to create futures in asyncio, as event loop implementations can provide alternative implementations of the Future class (with better performance or instrumentation).

New in version 3.5.2.

18.5.1.5. Tasks

`AbstractEventLoop.`**`create_task`**(*coro*)

Schedule the execution of a coroutine object: wrap it in a future. 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`.

This method was added in Python 3.4.2. Use the `async()` function to support also older Python versions.

New in version 3.4.2.

`AbstractEventLoop.`**`set_task_factory`**(*factory*)

Set a task factory that will be used by `AbstractEventLoop.create_task()`.

If *factory* is `None` the default task factory will be set.

If *factory* is a *callable*, it should have a signature matching `(loop, coro)`, where *loop* will be a reference to the active event loop, *coro* will be a coroutine object. The callable must return an `asyncio.Future` compatible object.

New in version 3.4.4.

`AbstractEventLoop.`**`get_task_factory`**()

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

New in version 3.4.4.

18.5.1.6. Creating connections

coroutine `AbstractEventLoop.`**`create_connection`**(*protocol_factory*, *host=None*, *port=None*, ***, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*)

Create a streaming transport connection to a given Internet *host* and *port*: socket family `AF_INET` or `AF_INET6` depending on *host* (or *family* if specified), socket type `SOCK_STREAM`. *protocol_factory* must be a callable returning a protocol instance.

This method is a coroutine which will try to establish the connection in the background. When successful, the coroutine 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 to represent it.
2. *protocol_factory* is called without arguments and must return a protocol instance.

3. The protocol instance is tied to the transport, and its `connection_made()` method is called.
4. The coroutine returns successfully with the `(transport, protocol)` pair.

The created transport is an implementation-dependent bidirectional stream.

Note: *protocol_factory* can be any kind of callable, not necessarily a class. For example, if you want to use a pre-created protocol instance, you can pass `lambda: my_protocol`.

Options that change how the connection is created:

- *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 context with some unspecified default settings is used.

  See also: SSL/TLS security considerations

- *server_hostname*, is only for use together with *ssl*, and sets or overrides the hostname that the target server's certificate will be matched against. 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 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*.

Changed in version 3.5: On Windows with `ProactorEventLoop`, SSL/TLS is now supported.

See also: The `open_connection()` function can be used to get a pair of (`StreamReader`, `StreamWriter`) instead of a protocol.

coroutine `AbstractEventLoop.`**`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 datagram connection: socket family `AF_INET` or `AF_INET6` depending on *host* (or *family* if specified), socket type `SOCK_DGRAM`. *protocol_factory* must be a callable returning a protocol instance.

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

Options changing how the connection is created:

- *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 UNIX's. 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 `AbstractEventLoop.`**`create_unix_connection`**(*protocol_factory*, *path*, ***, *ssl=None*, *sock=None*, *server_hostname=None*)

Create UNIX connection: socket family `AF_UNIX`, socket type `SOCK_STREAM`. The `AF_UNIX` socket family is used to communicate between processes on the same machine efficiently.

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

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

See the `AbstractEventLoop.create_connection()` method for parameters.

Availability: UNIX.

18.5.1.7. Creating listening connections

coroutine `AbstractEventLoop.`**`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*)

Create a TCP server (socket type `SOCK_STREAM`) bound to *host* and *port*.

Return a `Server` object, its `sockets` attribute contains created sockets. Use the `Server.close()` method to stop the server: close listening sockets.

Parameters:

- The *host* parameter can be a string, in that case the TCP server is bound to *host* and *port*. The *host* parameter can also be a sequence of strings and in that case the TCP server is bound to all hosts of 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 it will be determined from host (defaults to `socket.AF_UNSPEC`).
- *flags* is a bitmask for `getaddrinfo()`.
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* should be omitted (must be `None`).
- *backlog* is the maximum number of queued connections passed to `listen()` (defaults to 100).
- *ssl* can be set to an `SSLContext` to enable SSL 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.

This method is a coroutine.

Changed in version 3.5: On Windows with `ProactorEventLoop`, SSL/TLS is now supported.

> **See also:** The function `start_server()` creates a (`StreamReader`, `StreamWriter`) pair and calls back a function with this pair.

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

coroutine `AbstractEventLoop.`**`create_unix_server`**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*)

Similar to `AbstractEventLoop.create_server()`, but specific to the socket family `AF_UNIX`.

This method is a coroutine.

Availability: UNIX.

coroutine `BaseEventLoop.`**`connect_accepted_socket`**(*protocol_factory*, *sock*, ***, *ssl=None*)

Handle an accepted connection.

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

Parameters:

- *sock* is a preexisting socket object returned from an `accept` call.
- *ssl* can be set to an `SSLContext` to enable SSL over the accepted connections.

This method is a coroutine. When completed, the coroutine returns a `(transport, protocol)` pair.

New in version 3.5.3.

18.5.1.8. Watch file descriptors

On Windows with `SelectorEventLoop`, only socket handles are supported (ex: pipe file descriptors are not supported).

On Windows with `ProactorEventLoop`, these methods are not supported.

`AbstractEventLoop.`**`add_reader`**(*fd*, *callback*, **args*)

Start watching the file descriptor for read availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

`AbstractEventLoop.`**`remove_reader`**(*fd*)

Stop watching the file descriptor for read availability.

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

Start watching the file descriptor for write availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

`AbstractEventLoop.`**`remove_writer`**(*fd*)

Stop watching the file descriptor for write availability.

The watch a file descriptor for read events example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

18.5.1.9. Low-level socket operations

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

Receive data from the socket. Modeled after blocking `socket.socket.recv()` method.

The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *nbytes*.

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

This method is a coroutine.

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

Send data to the socket. Modeled after blocking `socket.socket.sendall()` method.

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

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

This method is a coroutine.

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

Connect to a remote socket at *address*. Modeled after blocking `socket.socket.connect()` method.

With `SelectorEventLoop` event loop, the socket *sock* must be non-blocking.

This method is a coroutine.

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, `AbstractEventLoop.getaddrinfo()` will be used to resolve the *address*.

> **See also:** `AbstractEventLoop.create_connection()` and `asyncio.open_connection()`.

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

Accept a connection. Modeled after blocking `socket.socket.accept()`.

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.

The socket *sock* must be non-blocking.

This method is a coroutine.

> **See also:** `AbstractEventLoop.create_server()` and `start_server()`.

18.5.1.10. Resolve host name

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

This method is a coroutine, similar to `socket.getaddrinfo()` function but non-blocking.

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

This method is a coroutine, similar to `socket.getnameinfo()` function but non-blocking.

18.5.1.11. Connect pipes

On Windows with `SelectorEventLoop`, these methods are not supported. Use `ProactorEventLoop` to support pipes on Windows.

coroutine `AbstractEventLoop.`**`connect_read_pipe`**(*protocol_factory*, *pipe*)

Register read pipe in eventloop.

protocol_factory should instantiate object with `Protocol` interface. *pipe* is a file-like object. Return pair `(transport, protocol)`, where *transport* supports the `ReadTransport` interface.

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

This method is a coroutine.

coroutine `AbstractEventLoop.`**`connect_write_pipe`**(*protocol_factory*, *pipe*)

Register write pipe in eventloop.

protocol_factory should instantiate object with `BaseProtocol` interface. *pipe* is file-like object. Return pair `(transport, protocol)`, where *transport* supports `WriteTransport` interface.

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

This method is a coroutine.

See also: The `AbstractEventLoop.subprocess_exec()` and `AbstractEventLoop.subprocess_shell()` methods.

18.5.1.12. UNIX signals

Availability: UNIX only.

`AbstractEventLoop.`**`add_signal_handler`**(*signum*, *callback*, **args*)

Add a handler for a signal.

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 keywords to the callback.

`AbstractEventLoop.`**`remove_signal_handler`**(*sig*)

Remove a handler for a signal.

Return `True` if a signal handler was removed, `False` if not.

See also: The `signal` module.

18.5.1.13. Executor

Call a function in an `Executor` (pool of threads or pool of processes). By default, an event loop uses a thread pool executor (`ThreadPoolExecutor`).

coroutine `AbstractEventLoop.`**`run_in_executor`**(*executor*, *func*, **args*)

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

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

Use functools.partial to pass keywords to the *func*.

This method is a coroutine.

Changed in version 3.5.3: `BaseEventLoop.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.

`AbstractEventLoop.`**`set_default_executor`**(*executor*)

Set the default executor used by `run_in_executor()`.

18.5.1.14. Error Handling API

Allows customizing how exceptions are handled in the event loop.

AbstractEventLoop.**set_exception_handler**(*handler*)

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

If *handler* is `None`, the default exception handler will be set.

If *handler* is a callable object, it should have a matching signature to `(loop, context)`, where `loop` will be a reference to the active event loop, `context` will be a `dict` object (see `call_exception_handler()` documentation for details about context).

AbstractEventLoop.**get_exception_handler**()

Return the exception handler, or `None` if the default one is in use.

New in version 3.5.2.

AbstractEventLoop.**default_exception_handler**(*context*)

Default exception handler.

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

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

AbstractEventLoop.**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 later):

- '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:** Note: this method should not be overloaded in subclassed event loops. For any custom exception handling, use `set_exception_handler()` method.

18.5.1.15. Debug mode

AbstractEventLoop.**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.

New in version 3.4.2.

`AbstractEventLoop.`**`set_debug`**(*enabled: bool*)

Set the debug mode of the event loop.

New in version 3.4.2.

See also: The debug mode of asyncio.

18.5.1.16. Server

class `asyncio.`**`Server`**

Server listening on sockets.

Object created by the `AbstractEventLoop.create_server()` method and the `start_server()` function. Don't instantiate the class directly.

`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.

coroutine **`wait_closed`**()

Wait until the `close()` method completes.

This method is a coroutine.

`sockets`

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

18.5.1.17. Handle

class `asyncio.`**`Handle`**

A callback wrapper object returned by AbstractEventLoop.call_soon(), AbstractEventLoop.call_soon_threadsafe(), AbstractEventLoop.call_later(), and AbstractEventLoop.call_at().

cancel()

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

18.5.1.18. Event loop examples

18.5.1.18.1. Hello World with call_soon()

Example using the AbstractEventLoop.call_soon() method to schedule a callback. The callback displays "Hello World" and then stops the event loop:

```
import asyncio

def hello_world(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()
loop.run_forever()
loop.close()
```

See also: The Hello World coroutine example uses a coroutine.

18.5.1.18.2. Display the current date with call_later()

Example of callback displaying the current date every second. The callback uses the AbstractEventLoop.call_later() method to reschedule itself during 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()
loop.run_forever()
loop.close()
```

See also: The coroutine displaying the current date example uses a coroutine.

18.5.1.18.3. Watch a file descriptor for read events

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

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils 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())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

See also: The register an open socket to wait for data using a protocol example uses a low-level protocol created by the `AbstractEventLoop.create_connection()` method.

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.

18.5.1.18.4. Set signal handlers for SIGINT and SIGTERM

Register handlers for signals `SIGINT` and `SIGTERM` using the `AbstractEventLoop.add_signal_handler()` method:

```
import asyncio
import functools
import os
import signal

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

loop = asyncio.get_event_loop()
for signame in ('SIGINT', 'SIGTERM'):
    loop.add_signal_handler(getattr(signal, signame),
                            functools.partial(ask_exit, signame))

print("Event loop running forever, press Ctrl+C to interrupt.")
print("pid %s: send SIGINT or SIGTERM to exit." % os.getpid())
try:
    loop.run_forever()
finally:
    loop.close()
```

This example only works on UNIX.