CHAPTER

NINETEEN

NETWORKING AND INTERPROCESS COMMUNICATION

The modules described in this chapter provide mechanisms for networking and inter-processes communication.

Some modules only work for two processes that are on the same machine, e.g. *signal* and *mmap*. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

19.1 asyncio — Asynchronous I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

# Python 3.7+
asyncio.run(main())
```

asyncio is a library to write **concurrent** code using the **async/await** syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level **structured** network code.

asyncio provides a set of **high-level** APIs to:

- *run Python coroutines* concurrently and have full control over their execution;
- perform *network IO and IPC*;
- control *subprocesses*;
- distribute tasks via *queues*;
- *synchronize* concurrent code;

Additionally, there are **low-level** APIs for *library and framework developers* to:

- create and manage *event loops*, which provide asynchronous APIs for *networking*, running *subprocesses*, handling *OS signals*, etc;
- implement efficient protocols using *transports*;
- *bridge* callback-based libraries and code with async/await syntax.

Reference

19.1.1 Coroutines and Tasks

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

Coroutines

Coroutines declared with async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code (requires Python 3.7+) prints "hello", waits 1 second, and then prints "world":

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine asyncio provides three main mechanisms:

- The *asyncio.run()* function to run the top-level entry point "main()" function (see the above example.)
- Awaiting on a coroutine. The following snippet of code will print "hello" after waiting for 1 second, and then print "world" after waiting for *another* 2 seconds:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- The *asyncio.create_task()* function to run coroutines concurrently as asyncio *Tasks*.

  Let's modify the above example and run two `say_after` coroutines *concurrently*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

  Note that expected output now shows that the snippet runs 1 second faster than before:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of *awaitable* objects: **coroutines**, **Tasks**, and **Futures**.

Coroutines

Python coroutines are *awaitables* and therefore can be awaited from other coroutines:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Important: In this documentation the term "coroutine" can be used for two closely related concepts:

- a *coroutine function*: an `async def` function;
- a *coroutine object*: an object returned by calling a *coroutine function*.

asyncio also supports legacy *generator-based* coroutines.

Tasks

Tasks are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like *asyncio.create_task()* the coroutine is automatically scheduled to run soon:

```
import asyncio

async def nested():
    return 42
```

(continues on next page)

(continued from previous page)

```
async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A *Future* is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.

When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

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

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is *loop.run_in_executor()*.

Running an asyncio Program

asyncio.**run**(*coro*, ***, *debug=False*)

This function runs the passed coroutine, taking care of managing the asyncio event loop and *finalizing asynchronous generators*.

This function cannot be called when another asyncio event loop is running in the same thread.

If *debug* is `True`, the event loop will be run in debug mode.

This function always creates a new event loop and closes it at the end. It should be used as a main entry point for asyncio programs, and should ideally only be called once.

New in version 3.7: **Important:** this function has been added to asyncio in Python 3.7 on a *provisional basis*.

Creating Tasks

asyncio.**create_task**(*coro*)

Wrap the *coro* *coroutine* into a *Task* and schedule its execution. Return the Task object.

The task is executed in the loop returned by *get_running_loop()*, *RuntimeError* is raised if there is no running loop in current thread.

This function has been **added in Python 3.7**. Prior to Python 3.7, the low-level *asyncio.ensure_future()* function can be used instead:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
task = asyncio.ensure_future(coro())
...
```

New in version 3.7.

Sleeping

coroutine asyncio.sleep(*delay*, *result=None*, *, *loop=None*)

Block for *delay* seconds.

If *result* is provided, it is returned to the caller when the coroutine completes.

`sleep()` always suspends the current task, allowing other tasks to run.

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

Example of coroutine displaying the current date every second for 5 seconds:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Running Tasks Concurrently

awaitable asyncio.gather(**aws*, *loop=None*, *return_exceptions=False*)

Run *awaitable objects* in the *aws* sequence *concurrently*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in *aws*.

If *return_exceptions* is `False` (default), the first raised exception is immediately propagated to the task that awaits on `gather()`. Other awaitables in the *aws* sequence **won't be cancelled** and will continue to run.

If *return_exceptions* is `True`, exceptions are treated the same as successful results, and aggregated in the result list.

If `gather()` is *cancelled*, all submitted awaitables (that have not completed yet) are also *cancelled*.

If any Task or Future from the *aws* sequence is *cancelled*, it is treated as if it raised *CancelledError* – the `gather()` call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

Example:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4)...
#     Task C: factorial(4) = 24
```

Changed in version 3.7: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

Shielding From Cancellation

awaitable asyncio.shield(*aw*, ***, *loop=None*)

Protect an *awaitable object* from being *cancelled*.

If *aw* is a coroutine it is automatically scheduled as a Task.

The statement:

```
res = await shield(something())
```

is equivalent to:

```
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the "await" expression still raises a *CancelledError*.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a try/except clause, as follows:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

Timeouts

coroutine asyncio.wait_for(*aw*, *timeout*, ***, *loop=None*)

Wait for the *aw* *awaitable* to complete with a timeout.

If *aw* is a coroutine it is automatically scheduled as a Task.

timeout can either be `None` or a float or int number of seconds to wait for. If *timeout* is `None`, block until the future completes.

If a timeout occurs, it cancels the task and raises *asyncio.TimeoutError*.

To avoid the task *cancellation*, wrap it in *shield()*.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*.

If the wait is cancelled, the future *aw* is also cancelled.

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

Example:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())
```

(continues on next page)

(continued from previous page)

```
# Expected output:
#
#     timeout!
```

Changed in version 3.7: When *aw* is cancelled due to a timeout, `wait_for` waits for *aw* to be cancelled. Previously, it raised *asyncio.TimeoutError* immediately.

Waiting Primitives

coroutine asyncio.wait(*aws*, ***, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Run *awaitable objects* in the *aws* set concurrently and block until the condition specified by *return_when*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task. Passing coroutines objects to `wait()` directly is deprecated as it leads to *confusing behavior*.

Returns two sets of Tasks/Futures: `(done, pending)`.

Usage:

```
done, pending = await asyncio.wait(aws)
```

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

timeout (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise *asyncio.TimeoutError*. Futures or Tasks that aren't done when the timeout occurs are simply returned in the second set.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
`FIRST_COMPLETED`	The function will return when any future finishes or is cancelled.
`FIRST_EXCEPTION`	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
`ALL_COMPLETED`	The function will return when all futures finish or are cancelled.

Unlike *wait_for()*, `wait()` does not cancel the futures when a timeout occurs.

Note: `wait()` schedules coroutines as Tasks automatically and later returns those implicitly created Task objects in `(done, pending)` sets. Therefore the following code won't work as expected:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

Here is how the above snippet can be fixed:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

Passing coroutine objects to `wait()` directly is deprecated.

---

asyncio.**as_completed**(*aws*, ***, *loop=None*, *timeout=None*)

Run *awaitable objects* in the *aws* set concurrently. Return an iterator of *Future* objects. Each Future object returned represents the earliest result from the set of the remaining awaitables.

Raises *asyncio.TimeoutError* if the timeout occurs before all Futures are done.

Example:

```
for f in as_completed(aws):
    earliest_result = await f
    # ...
```

Scheduling From Other Threads

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)

Submit a coroutine to the given event loop. Thread-safe.

Return a *concurrent.futures.Future* to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

See the *concurrency and multithreading* section of the documentation.

Unlike other asyncio functions this function requires the *loop* argument to be passed explicitly.

New in version 3.5.1.

Introspection

asyncio.**current_task**(*loop=None*)
Return the currently running *Task* instance, or `None` if no task is running.

If *loop* is `None` *get_running_loop()* is used to get the current loop.

New in version 3.7.

asyncio.**all_tasks**(*loop=None*)
Return a set of not yet finished *Task* objects run by the loop.

If *loop* is `None`, *get_running_loop()* is used for getting current loop.

New in version 3.7.

Task Object

class asyncio.**Task**(*coro*, ***, *loop=None*)
A *Future-like* object that runs a Python *coroutine*. Not thread-safe.

Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level *asyncio.create_task()* function to create Tasks, or the low-level *loop.create_task()* or *ensure_future()* functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the *cancel()* method. Calling it will cause the Task to throw a *CancelledError* exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

cancelled() can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the *CancelledError* exception and was actually cancelled.

asyncio.Task inherits from *Future* all of its APIs except *Future.set_result()* and *Future.set_exception()*.

Tasks support the *contextvars* module. When a Task is created it copies the current context and later runs its coroutine in the copied context.

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

cancel()
Request the Task to be cancelled.

This arranges for a *CancelledError* exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a `try` … … `except CancelledError` … `finally` block. Therefore, unlike *Future.cancel()*, *Task.cancel()* does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged.

The following example illustrates how coroutines can intercept the cancellation request:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()
Return `True` if the Task is *cancelled*.

The Task is *cancelled* when the cancellation was requested with *cancel()* and the wrapped coroutine propagated the *CancelledError* exception thrown into it.

done()
Return `True` if the Task is *done*.

A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

result()
Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

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

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

exception()
Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

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

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

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

This method should only be used in low-level callback-based code.

See the documentation of *Future.add_done_callback()* for more details.

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

This method should only be used in low-level callback-based code.

See the documentation of *Future.remove_done_callback()* for more details.

get_stack(*, *limit=None*)
Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional *limit* argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

print_stack(*, *limit=None*, *file=None*)
Print the stack or traceback for this Task.

This produces output similar to that of the traceback module for the frames retrieved by *get_stack()*.

The *limit* argument is passed to *get_stack()* directly.

The *file* argument is an I/O stream to which the output is written; by default output is written to *sys.stderr*.

classmethod all_tasks(*loop=None*)
Return a set of all tasks for an event loop.

By default all tasks for the current event loop are returned. If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the *asyncio.all_tasks()* function instead.

classmethod current_task(*loop=None*)
Return the currently running task or `None`.

If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the *asyncio.current_task()* function instead.

Generator-based Coroutines

Note: Support for generator-based coroutines is **deprecated** and is scheduled for removal in Python 3.10.

Generator-based coroutines predate async/await syntax. They are Python generators that use `yield from` expressions to await on Futures and other coroutines.

Generator-based coroutines should be decorated with *@asyncio.coroutine*, although this is not enforced.

`@asyncio.coroutine`
Decorator to mark generator-based coroutines.

This decorator enables legacy generator-based coroutines to be compatible with async/await code:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

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

This decorator is **deprecated** and is scheduled for removal in Python 3.10.

This decorator should not be used for `async def` coroutines.

`asyncio.iscoroutine(`*obj*`)`
Return `True` if *obj* is a *coroutine object*.

This method is different from *inspect.iscoroutine()* because it returns `True` for generator-based coroutines.

`asyncio.iscoroutinefunction(`*func*`)`
Return `True` if *func* is a *coroutine function*.

This method is different from *inspect.iscoroutinefunction()* because it returns `True` for generator-based coroutine functions decorated with *@coroutine*.

19.1.2 Streams

Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.

Here is an example of a TCP echo client written using asyncio streams:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
```

(continues on next page)

(continued from previous page)

```
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also the *Examples* section below.

Stream Functions

The following top-level asyncio functions can be used to create and work with streams:

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Establish a network connection and return a pair of `(reader, writer)` objects.

The returned *reader* and *writer* objects are instances of *StreamReader* and *StreamWriter* classes.

The *loop* argument is optional and can always be determined automatically when this function is awaited from a coroutine.

limit determines the buffer size limit used by the returned *StreamReader* instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to *loop.create_connection()*.

New in version 3.7: The *ssl_handshake_timeout* parameter.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *loop=None*, *limit=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*)

Start a socket server.

The *client_connected_cb* callback is called whenever a new client connection is established. It receives a `(reader, writer)` pair as two arguments, instances of the *StreamReader* and *StreamWriter* classes.

client_connected_cb can be a plain callable or a *coroutine function*; if it is a coroutine function, it will be automatically scheduled as a *Task*.

The *loop* argument is optional and can always be determined automatically when this method is awaited from a coroutine.

limit determines the buffer size limit used by the returned *StreamReader* instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to *loop.create_server()*.

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

Unix Sockets

coroutine asyncio.**open_unix_connection**(*path=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Establish a Unix socket connection and return a pair of `(reader, writer)`.

Similar to *open_connection()* but operates on Unix sockets.

See also the documentation of *loop.create_unix_connection()*.

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*

coroutine asyncio.start_unix_server(*client_connected_cb*, *path=None*, ***, *loop=None*, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Start a Unix socket server.

Similar to *start_server()* but works with Unix sockets.

See also the documentation of *loop.create_unix_server()*.

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-like object*.

StreamReader

class asyncio.StreamReader

Represents a reader object that provides APIs to read data from the IO stream.

It is not recommended to instantiate *StreamReader* objects directly; use *open_connection()* and *start_server()* instead.

coroutine read(*n=-1*)

Read up to *n* bytes. If *n* is not provided, or set to `-1`, read until EOF and return all read bytes.

If EOF was received and the internal buffer is empty, return an empty `bytes` object.

coroutine readline()

Read one line, where "line" is a sequence of bytes ending with `\n`.

If EOF is received and `\n` was not found, the method returns partially read data.

If EOF is received and the internal buffer is empty, return an empty `bytes` object.

coroutine readexactly(*n*)

Read exactly *n* bytes.

Raise an *IncompleteReadError* if EOF is reached before *n* can be read. Use the *IncompleteReadError.partial* attribute to get the partially read data.

coroutine readuntil(*separator=b'\n'*)

Read data from the stream until *separator* is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

If the amount of data read exceeds the configured stream limit, a *LimitOverrunError* exception is raised, and the data is left in the internal buffer and can be read again.

If EOF is reached before the complete separator is found, an *IncompleteReadError* exception is raised, and the internal buffer is reset. The *IncompleteReadError.partial* attribute may contain a portion of the separator.

New in version 3.5.2.

at_eof()

Return `True` if the buffer is empty and `feed_eof()` was called.

StreamWriter

class asyncio.StreamWriter

Represents a writer object that provides APIs to write data to the IO stream.

It is not recommended to instantiate *StreamWriter* objects directly; use *open_connection()* and *start_server()* instead.

can_write_eof()

Return *True* if the underlying transport supports the *write_eof()* method, *False* otherwise.

write_eof()

Close the write end of the stream after the buffered write data is flushed.

transport

Return the underlying asyncio transport.

get_extra_info(*name*, *default=None*)

Access optional transport information; see *BaseTransport.get_extra_info()* for details.

write(*data*)

Write *data* to the stream.

This method is not subject to flow control. Calls to `write()` should be followed by *drain()*.

writelines(*data*)

Write a list (or any iterable) of bytes to the stream.

This method is not subject to flow control. Calls to `writelines()` should be followed by *drain()*.

coroutine drain()

Wait until it is appropriate to resume writing to the stream. Example:

```
writer.write(data)
await writer.drain()
```

This is a flow control method that interacts with the underlying IO write buffer. When the size of the buffer reaches the high watermark, *drain()* blocks until the size of the buffer is drained down to the low watermark and writing can be resumed. When there is nothing to wait for, the *drain()* returns immediately.

close()

Close the stream.

is_closing()

Return `True` if the stream is closed or in the process of being closed.

New in version 3.7.

coroutine wait_closed()

Wait until the stream is closed.

Should be called after *close()* to wait until the underlying connection is closed.

New in version 3.7.

Examples

TCP echo client using streams

TCP echo client using the *asyncio.open_connection()* function:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also:

The *TCP echo client protocol* example uses the low-level *loop.create_connection()* method.

TCP echo server using streams

TCP echo server using the *asyncio.start_server()* function:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

See also:

The *TCP echo server protocol* example uses the *loop.create_server()* method.

Get HTTP headers

Simple example querying HTTP headers of the URL passed on the command line:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Usage:

```
python example.py http://example.com/path/page.html
```

or with HTTPS:

```
python example.py https://example.com/path/page.html
```

Register an open socket to wait for data using streams

Coroutine waiting until a socket receives data using the *open_connection()* function:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

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

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

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

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

See also:

The *register an open socket to wait for data using a protocol* example uses a low-level protocol and the *loop.create_connection()* method.

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

19.1.3 Synchronization Primitives

asyncio synchronization primitives are designed to be similar to those of the *threading* module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use *threading* for that);
- methods of these synchronization primitives do not accept the *timeout* argument; use the *asyncio.wait_for()* function to perform operations with timeouts.

asyncio has the following basic sychronization primitives:

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*

Lock

class asyncio.Lock(*, *loop=None*)

Implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

coroutine acquire()

Acquire the lock.

This method waits until the lock is *unlocked*, sets it to *locked* and returns `True`.

release()

Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a *RuntimeError* is raised.

locked()

Return `True` if the lock is *locked*.

Event

class asyncio.Event(*, *loop=None*)

An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to *true* with the *set()* method and reset to *false* with the *clear()* method. The *wait()* method blocks until the flag is set to *true*. The flag is set to *false* initially.

Example:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()
Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls *set()*.

set()
Set the event.

All tasks waiting for event to be set will be immediately awakened.

clear()
Clear (unset) the event.

Tasks awaiting on *wait()* will now block until the *set()* method is called again.

is_set()
Return `True` if the event is set.

Condition

class asyncio.Condition(*lock=None*, ***, *loop=None*)
A Condition object. Not thread-safe.

An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an *Event* and a *Lock*. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional *lock* argument must be a *Lock* object or `None`. In the latter case a new Lock object is created automatically.

The preferred way to use a Condition is an `async with` statement:

```
cond = asyncio.Condition()
```

(continues on next page)

(continued from previous page)

```
# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```
cond = asyncio.Condition()

# ... later
await lock.acquire()
try:
    await cond.wait()
finally:
    lock.release()
```

coroutine acquire()
Acquire the underlying lock.

This method waits until the underlying lock is *unlocked*, sets it to *locked* and returns `True`.

notify(*n=1*)
Wake up at most *n* tasks (1 by default) waiting on this condition. The method is no-op if no tasks are waiting.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

locked()
Return `True` if the underlying lock is acquired.

notify_all()
Wake up all tasks waiting on this condition.

This method acts like *notify()*, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

release()
Release the underlying lock.

When invoked on an unlocked lock, a *RuntimeError* is raised.

coroutine wait()
Wait until notified.

If the calling task has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call. Once awakened, the Condition re-acquires its lock and this method returns `True`.

coroutine wait_for(*predicate*)
Wait until a predicate becomes *true*.

The predicate must be a callable which result will be interpreted as a boolean value. The final value is the return value.

Semaphore

class asyncio.Semaphore(*value=1*, *, *loop=None*)

A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some task calls *release()*.

The optional *value* argument gives the initial value for the internal counter (`1` by default). If the given value is less than `0` a *ValueError* is raised.

The preferred way to use a Semaphore is an `async with` statement:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return `True` immediately. If it is zero, wait until a *release()* is called and return `True`.

locked()

Returns `True` if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike *BoundedSemaphore*, *Semaphore* allows making more `release()` calls than `acquire()` calls.

BoundedSemaphore

class asyncio.BoundedSemaphore(*value=1*, *, *loop=None*)

A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of *Semaphore* that raises a *ValueError* in *release()* if it increases the internal counter above the initial *value*.

Deprecated since version 3.7: Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (`with await lock`, `with (yield from lock)`) is deprecated. Use `async with lock` instead.

19.1.4 Subprocesses

This section describes high-level async/await asyncio APIs to create and manage subprocesses.

Here's an example of how asyncio can run a shell command and obtain its result:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

will print:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel. It is indeed trivial to modify the above example to run several commands simultaneously:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

See also the *Examples* subsection.

Creating Subprocesses

coroutine asyncio.create_subprocess_exec(**args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Create a subprocess.

The *limit* argument sets the buffer limit for *StreamReader* wrappers for `Process.stdout` and `Process.stderr` (if *subprocess.PIPE* is passed to *stdout* and *stderr* arguments).

Return a *Process* instance.

See the documentation of *loop.subprocess_exec()* for other parameters.

coroutine asyncio.create_subprocess_shell(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Run the *cmd* shell command.

The *limit* argument sets the buffer limit for *StreamReader* wrappers for `Process.stdout` and `Process.stderr` (if *subprocess.PIPE* is passed to *stdout* and *stderr* arguments).

Return a *Process* instance.

See the documentation of *loop.subprocess_shell()* for other parameters.

Important: 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 shell characters in strings that are going to be used to construct shell commands.

Note: The default asyncio event loop implementation on **Windows** does not support subprocesses. Subprocesses are available for Windows if a *ProactorEventLoop* is used. See *Subprocess Support on Windows* for details.

See also:

asyncio also has the following *low-level* APIs to work with subprocesses: *loop.subprocess_exec()*, *loop.subprocess_shell()*, *loop.connect_read_pipe()*, *loop.connect_write_pipe()*, as well as the *Subprocess Transports* and *Subprocess Protocols*.

Constants

asyncio.subprocess.PIPE

Can be passed to the *stdin*, *stdout* or *stderr* parameters.

If *PIPE* is passed to *stdin* argument, the *Process.stdin* attribute will point to a *StreamWriter* instance.

If *PIPE* is passed to *stdout* or *stderr* arguments, the *Process.stdout* and *Process.stderr* attributes will point to *StreamReader* instances.

asyncio.subprocess.STDOUT

Special value that can be used as the *stderr* argument and indicates that standard error should be redirected into standard output.

asyncio.subprocess.DEVNULL

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to process creation functions. It indicates that the special file *os.devnull* will be used for the corresponding subprocess stream.

Interacting with Subprocesses

Both *create_subprocess_exec()* and *create_subprocess_shell()* functions return instances of the *Process* class. *Process* is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

class asyncio.subprocess.Process

An object that wraps OS processes created by the *create_subprocess_exec()* and *create_subprocess_shell()* functions.

This class is designed to have a similar API to the *subprocess.Popen* class, but there are some notable differences:

- unlike Popen, Process instances do not have an equivalent to the *poll()* method;
- the *communicate()* and *wait()* methods don't have a *timeout* parameter: use the *wait_for()* function;
- the *Process.wait()* method is asynchronous, whereas *subprocess.Popen.wait()* method is implemented as a blocking busy loop;
- the *universal_newlines* parameter is not supported.

This class is *not thread safe*.

See also the *Subprocess and Threads* section.

coroutine wait()
Wait for the child process to terminate.

Set and return the *returncode* attribute.

Note: This method can deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates so much output that it blocks waiting for the OS pipe buffer to accept more data. Use the *communicate()* method when using pipes to avoid this condition.

---

coroutine communicate(*input=None*)
Interact with process:

1. send data to *stdin* (if *input* is not `None`);
2. read data from *stdout* and *stderr*, until EOF is reached;
3. wait for process to terminate.

The optional *input* argument is the data (*bytes* object) that will be sent to the child process.

Return a tuple `(stdout_data, stderr_data)`.

If either *BrokenPipeError* or *ConnectionResetError* exception is raised when writing *input* into *stdin*, the exception is ignored. This condition occurs when the process exits before all data are written into *stdin*.

If it is desired to send data to the process' *stdin*, the process needs to be created with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, the process has to be created with `stdout=PIPE` and/or `stderr=PIPE` arguments.

Note, that the data read is buffered in memory, so do not use this method if the data size is large or unlimited.

send_signal(*signal*)
Sends the signal *signal* to the child process.

Note: On Windows, `SIGTERM` is an alias for *terminate()*. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

---

terminate()
Stop the child process.

On POSIX systems this method sends `signal.SIGTERM` to the child process.

On Windows the Win32 API function `TerminateProcess()` is called to stop the child process.

kill()
Kill the child.

On POSIX systems this method sends `SIGKILL` to the child process.

On Windows this method is an alias for *terminate()*.

stdin
Standard input stream (*StreamWriter*) or `None` if the process was created with `stdin=None`.

stdout
Standard output stream (*StreamReader*) or `None` if the process was created with `stdout=None`.

stderr
Standard error stream (*StreamReader*) or `None` if the process was created with `stderr=None`.

> **Warning:** Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid
Process identification number (PID).

Note that for processes created by the *create_subprocess_shell()* function, this attribute is the PID of the spawned shell.

returncode
Return code of the process when it exits.

A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

Subprocess and Threads

Standard asyncio event loop supports running subprocesses from different threads, but there are limitations:

- An event loop must run in the main thread.
- The child watcher must be instantiated in the main thread before executing subprocesses from other threads. Call the *get_child_watcher()* function in the main thread to instantiate the child watcher.

Note that alternative event loop implementations might not share the above limitations; please refer to their documentation.

See also:

The *Concurrency and multithreading in asyncio* section.

Examples

An example using the *Process* class to control a subprocess and the *StreamReader* class to read from its standard output.

The subprocess is created by the *create_subprocess_exec()* function:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'
```

(continues on next page)

(continued from previous page)

```
    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

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 low-level APIs.

19.1.5 Queues

asyncio queues are designed to be similar to classes of the *queue* module. Although asyncio queues are not thread-safe, they are designed to be used specifically in async/await code.

Note that methods of asyncio queues don't have a *timeout* parameter; use *asyncio.wait_for()* function to do queue operations with a timeout.

See also the *Examples* section below.

Queue

class asyncio.**Queue**(*maxsize=0*, ***, *loop=None*)

A first in, first out (FIFO) queue.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than `0`, then `await put()` blocks when the queue reaches *maxsize* until an item is removed by *get()*.

Unlike the standard library threading *queue*, the size of the queue is always known and can be returned by calling the *qsize()* method.

This class is *not thread safe*.

maxsize

Number of items allowed in the queue.

empty()

Return `True` if the queue is empty, `False` otherwise.

full()

Return `True` if there are *maxsize* items in the queue.

If the queue was initialized with `maxsize=0` (the default), then *full()* never returns `True`.

coroutine get()
Remove and return an item from the queue. If queue is empty, wait until an item is available.

get_nowait()
Return an item if one is immediately available, else raise *QueueEmpty*.

coroutine join()
Block until all items in the queue have been received and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

coroutine put(*item***)**
Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.

put_nowait(*item***)**
Put an item into the queue without blocking.

If no free slot is immediately available, raise *QueueFull*.

qsize()
Return the number of items in the queue.

task_done()
Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises *ValueError* if called more times than there were items placed in the queue.

Priority Queue

class asyncio.PriorityQueue
A variant of *Queue*; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form `(priority_number, data)`.

LIFO Queue

class asyncio.LifoQueue
A variant of *Queue* that retrieves most recently added entries first (last in, first out).

Exceptions

exception asyncio.QueueEmpty
This exception is raised when the *get_nowait()* method is called on an empty queue.

exception asyncio.QueueFull
Exception raised when the *put_nowait()* method is called on a queue that has reached its *maxsize*.

Examples

Queues can be used to distribute workload between several concurrent tasks:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
```

(continues on next page)

(continued from previous page)

```
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.6 Exceptions

exception asyncio.TimeoutError

The operation has exceeded the given deadline.

Important: This exception is different from the builtin *TimeoutError* exception.

---

exception asyncio.CancelledError

The operation has been cancelled.

This exception can be caught to perform custom operations when asyncio Tasks are cancelled. In almost all situations the exception must be re-raised.

Important: This exception is a subclass of *Exception*, so it can be accidentally suppressed by an overly broad `try..except` block:

```
try:
    await operation
except Exception:
    # The cancellation is broken because the *except* block
    # suppresses the CancelledError exception.
    log.log('an error has occurred')
```

Instead, the following pattern should be used:

```
try:
    await operation
except asyncio.CancelledError:
    raise
except Exception:
    log.log('an error has occurred')
```

---

exception asyncio.InvalidStateError

Invalid internal state of *Task* or *Future*.

Can be raised in situations like setting a result value for a *Future* object that already has a result value set.

exception asyncio.SendfileNotAvailableError

The "sendfile" syscall is not available for the given socket or file type.

A subclass of *RuntimeError*.

exception asyncio.IncompleteReadError

The requested read operation did not complete fully.

Raised by the *asyncio stream APIs*.

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.

- A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.
  - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For *AF_INET6* address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where *flowinfo* and *scopeid* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For *socket* module methods, *flowinfo* and *scopeid* can be omitted just for backward compatibility. Note, however, omission of *scopeid* can cause problems in manipulating scoped IPv6 addresses.

  Changed in version 3.7: For multicast addresses (with *scopeid* meaningful) *address* may not contain `%scope` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).
- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.
- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:
  - *addr_type* is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
  - *scope* is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
  - If *addr_type* is `TIPC_ADDR_NAME`, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is `TIPC_ADDR_NAMESEQ`, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is `TIPC_ADDR_ID`, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.
- A tuple `(interface, )` is used for the *AF_CAN* address family, where *interface* is a string representing a network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.
  - *CAN_ISOTP* protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  New in version 3.3.
- `AF_BLUETOOTH` supports the following protocols and address formats:
  - `BTPROTO_L2CAP` accepts `(bdaddr, psm)` where `bdaddr` is the Bluetooth address as a string and `psm` is an integer.
  - `BTPROTO_RFCOMM` accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
  - `BTPROTO_HCI` accepts `(device_id,)` where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

    Changed in version 3.2: NetBSD and DragonFlyBSD support added.
  - `BTPROTO_SCO` accepts `bdaddr` where `bdaddr` is a *bytes* object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.

- *AF_ALG* is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:
  - *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
  - *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
  - *feat* and *mask* are unsigned 32bit integers.

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  New in version 3.6.
- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

  New in version 3.7.
- *AF_PACKET* is a low-level interface directly to network devices. The packets are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
  - *ifname* - String specifying the device name.
  - *proto* - An in network-byte-order integer specifying the Ethernet protocol number.
  - *pkttype* - Optional integer specifying the packet type:
    - `PACKET_HOST` (the default) - Packet addressed to the local host.
    - `PACKET_BROADCAST` - Physical-layer broadcast packet.
    - `PACKET_MULTIHOST` - Packet sent to a physical-layer multicast address.
    - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
  - *hatype* - Optional integer specifying the ARP hardware address type.
  - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise *OSError* or one of its subclasses (they used to raise *socket.error*).

Non-blocking mode is supported through *setblocking()*. A generalization of this based on timeouts is supported through *settimeout()*.

19.2.2 Module contents

The module *socket* exports the following elements.

Exceptions

exception socket.error
A deprecated alias of *OSError*.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of *OSError*.

exception socket.herror
A subclass of *OSError*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *gethostbyname_ex()* and *gethostbyaddr()*. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

Changed in version 3.3: This class was made a subclass of *OSError*.

exception socket.gaierror
A subclass of *OSError*, this exception is raised for address-related errors by *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

Changed in version 3.3: This class was made a subclass of *OSError*.

exception socket.timeout
A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always "timed out".

Changed in version 3.3: This class was made a subclass of *OSError*.

Constants

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` *IntEnum* collections.

New in version 3.4.

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6
These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET
These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.SOCK_CLOEXEC
socket.SOCK_NONBLOCK
These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

See also:

Secure File Descriptor Handling for a more thorough explanation.

Availability: Linux >= 2.6.27.

New in version 3.2.

SO_*
socket.SOMAXCONN
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

Changed in version 3.6: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

Changed in version 3.6.5: On Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` appear if run-time Windows supports.

Changed in version 3.7: `TCP_NOTSENT_LOWAT` was added.

On Windows, `TCP_KEEPIDLE`, `TCP_KEEPINTVL` appear if run-time Windows supports.

socket.AF_CAN
socket.PF_CAN
SOL_CAN_*
CAN_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

New in version 3.3.

socket.CAN_BCM
CAN_BCM_*

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

New in version 3.4.

socket.CAN_RAW_FD_FRAMES

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you one must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability: Linux >= 3.6.

New in version 3.5.

socket.CAN_ISOTP
: CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

  Availability: Linux >= 2.6.25.

  New in version 3.7.

socket.AF_PACKET
socket.PF_PACKET
PACKET_*
: Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

  Availability: Linux >= 2.2.

socket.AF_RDS
socket.PF_RDS
socket.SOL_RDS
RDS_*
: Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

  Availability: Linux >= 2.6.30.

  New in version 3.3.

socket.SIO_RCVALL
socket.SIO_KEEPALIVE_VALS
socket.SIO_LOOPBACK_FAST_PATH
RCVALL_*
: Constants for Windows' WSAIoctl(). The constants are used as arguments to the *ioctl()* method of socket objects.

  Changed in version 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

TIPC_*
: TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.AF_ALG
socket.SOL_ALG
ALG_*
: Constants for Linux Kernel cryptography.

  Availability: Linux >= 2.6.38.

  New in version 3.6.

socket.AF_VSOCK
socket.IOCTL_VM_SOCKETS_GET_LOCAL_CID
VMADDR*
SO_VM*
: Constants for Linux host/guest communication.

  Availability: Linux >= 4.8.

  New in version 3.7.

socket.AF_LINK
: *Availability*: BSD, OSX.

  New in version 3.4.

socket.has_ipv6
: This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.BDADDR_ANY
socket.BDADDR_LOCAL
: These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with BTPROTO_RFCOMM.

socket.HCI_FILTER
socket.HCI_TIME_STAMP
socket.HCI_DATA_DIR
: For use with BTPROTO_HCI. *HCI_FILTER* is not available for NetBSD or DragonFlyBSD. *HCI_TIME_STAMP* and *HCI_DATA_DIR* are not available for FreeBSD, NetBSD, or DragonFlyBSD.

Functions

Creating sockets

The following functions all create *socket objects*.

socket.socket(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)
: Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of CAN_RAW, *CAN_BCM* or *CAN_ISOTP*.

  If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

  The newly created socket is *non-inheritable*.

  Changed in version 3.3: The AF_CAN family was added. The AF_RDS family was added.

  Changed in version 3.4: The CAN_BCM protocol was added.

  Changed in version 3.4: The returned socket is now non-inheritable.

  Changed in version 3.7: The CAN_ISOTP protocol was added.

  Changed in version 3.7: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system *socket()* call. Therefore::

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM | socket.SOCK_NONBLOCK)

  will still create a non-blocking socket on OSes that support SOCK_NONBLOCK, but sock.type will be set to socket.SOCK_STREAM.

socket.socketpair([*family*[, *type*[, *proto*]]])
: Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the *socket()* function above. The default family is *AF_UNIX* if defined on the platform; otherwise, the default is *AF_INET*.

  The newly created sockets are *non-inheritable*.

Changed in version 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Changed in version 3.4: The returned sockets are now non-inheritable.

Changed in version 3.5: Windows support added.

`socket.create_connection`(*address*[, *timeout*[, *source_address*]])
Connect to a TCP service listening on the Internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than *socket.connect()*: if *host* is a non-numeric hostname, it will try to resolve it for both *AF_INET* and *AF_INET6*, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by *getdefaulttimeout()* is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

Changed in version 3.2: *source_address* was added.

`socket.fromfd`(*fd*, *family*, *type*, *proto=0*)
Duplicate the file descriptor *fd* (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the *socket()* function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The returned socket is now non-inheritable.

`socket.fromshare`(*data*)
Instantiate a socket from data obtained from the *socket.share()* method. The socket is assumed to be in blocking mode.

Availability: Windows.

New in version 3.3.

`socket.SocketType`
This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Other functions

The *socket* module also offers various network-related services:

`socket.close`(*fd*)
Close a socket file descriptor. This is like *os.close()*, but for sockets. On some platforms (most noticeable Windows) *os.close()* does not work for socket file descriptors.

New in version 3.7.

`socket.getaddrinfo`(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
Translate the *host/port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an

IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

`(family, type, proto, canonname, sockaddr)`

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the *socket()* function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for *AF_INET*, a `(address, port, flow info, scope id)` 4-tuple for *AF_INET6*), and is meant to be passed to the *socket.connect()* method.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

Changed in version 3.2: parameters can now be passed using keyword arguments.

Changed in version 3.7: for IPv6 multicast addresses, string representing an address will not contain `%scope` part.

socket.getfqdn([*name*])
Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by *gethostbyaddr()* is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by *gethostname()* is returned.

socket.gethostbyname(*hostname*)
Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See *gethostbyname_ex()* for a more complete interface. *gethostbyname()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

socket.gethostbyname_ex(*hostname*)
Translate a host name to IPv4 address format, extended interface. Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). *gethostbyname_ex()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

socket.gethostname()
Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Note: *gethostname()* doesn't always return the fully qualified domain name; use *getfqdn()* for that.

socket.**gethostbyaddr**(*ip_address*)
Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function *getfqdn()*. *gethostbyaddr()* supports both IPv4 and IPv6.

socket.**getnameinfo**(*sockaddr*, *flags*)
Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully-qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope` is appended to the host part if *sockaddr* contains meaningful *scopeid*. Usually this happens for multicast addresses.

socket.**getprotobyname**(*protocolname*)
Translate an Internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the *socket()* function. This is usually only needed for sockets opened in "raw" mode (*SOCK_RAW*); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

socket.**getservbyname**(*servicename*[, *protocolname*])
Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

socket.**getservbyport**(*port*[, *protocolname*])
Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

socket.**ntohl**(*x*)
Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**ntohs**(*x*)
Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Deprecated since version 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

socket.**htonl**(*x*)
Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**htons**(*x*)
Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Deprecated since version 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

socket.**inet_aton**(*ip_string*)
Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page *inet(3)* for details.

If the IPv4 address string passed to this function is invalid, *OSError* will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

inet_aton() does not support IPv6, and *inet_pton()* should be used instead for IPv4/v6 dual stack support.

`socket.inet_ntoa(`*packed_ip*`)`
Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, *OSError* will be raised. *inet_ntoa()* does not support IPv6, and *inet_ntop()* should be used instead for IPv4/v6 dual stack support.

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

`socket.inet_pton(`*address_family*, *ip_string*`)`
Convert an IP address from its family-specific string format to a packed, binary format. *inet_pton()* is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to *inet_aton()*) or `struct in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the IP address string *ip_string* is invalid, *OSError* will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added

`socket.inet_ntop(`*address_family*, *packed_ip*`)`
Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). *inet_ntop()* is useful when a library or network protocol returns an object of type `struct in_addr` (similar to *inet_ntoa()*) or `struct in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the bytes object *packed_ip* is not the correct length for the specified address family, *ValueError* will be raised. *OSError* is raised for errors from the call to *inet_ntop()*.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added

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

`socket.CMSG_LEN(`*length*`)`
Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for *recvmsg()* to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use *CMSG_SPACE()* and thus include space for padding, even when the item will be the last in the buffer. Raises *OverflowError* if *length* is outside the permissible range of values.

Availability: most Unix platforms, possibly others.

New in version 3.3.

`socket.CMSG_SPACE(`*length*`)`
Return the buffer size needed for *recvmsg()* to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the *CMSG_SPACE()* values for their associated data lengths. Raises *OverflowError* if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: most Unix platforms, possibly others.

New in version 3.3.

`socket.getdefaulttimeout()`
Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

`socket.setdefaulttimeout(`*timeout*`)`
Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See *settimeout()* for possible values and their respective meanings.

`socket.sethostname(`*name*`)`
Set the machine's hostname to *name*. This will raise an *OSError* if you don't have enough rights.

Availability: Unix.

New in version 3.3.

`socket.if_nameindex()`
Return a list of network interface information (index int, name string) tuples. *OSError* if the system call fails.

Availability: Unix.

New in version 3.3.

`socket.if_nametoindex(`*if_name*`)`
Return a network interface index number corresponding to an interface name. *OSError* if no interface with the given name exists.

Availability: Unix.

New in version 3.3.

`socket.if_indextoname(`*if_index*`)`
Return a network interface name corresponding to an interface index number. *OSError* if no interface with the given index exists.

Availability: Unix.

New in version 3.3.

19.2.3 Socket Objects

Socket objects have the following methods. Except for *makefile()*, these correspond to Unix system calls applicable to sockets.

Changed in version 3.2: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *close()*.

`socket.accept()`
Accept a connection. 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 newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

`socket.bind(`*address*`)`
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

`socket.close()`
Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from *makefile()* are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly, or to use a `with` statement around them.

Changed in version 3.6: *OSError* is now raised if an error occurs when the underlying `close()` call is made.

Note: *close()* releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call *shutdown()* before *close()*.

`socket.connect(`*address*`)`
Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a *socket.timeout* on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an *InterruptedError* exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Changed in version 3.5: The method now waits until the connection completes instead of raising an *InterruptedError* exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** for the rationale).

`socket.connect_ex(`*address*`)`
Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

`socket.detach()`
Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

New in version 3.2.

`socket.dup()`
Duplicate the socket.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

`socket.fileno()`
Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with *select.select()*.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as *os.fdopen()*). Unix does not have this limitation.

socket.get_inheritable()
Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

New in version 3.4.

socket.getpeername()
Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.getsockname()
Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.getsockopt(*level*, *optname*[, *buflen*])
Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (`SO_*` etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as byte strings).

socket.getblocking()
Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() == 0`.

New in version 3.7.

socket.gettimeout()
Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to *setblocking()* or *settimeout()*.

socket.ioctl(*control*, *option*)

Platform Windows

The *ioctl()* method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic *fcntl.fcntl()* and *fcntl.ioctl()* functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Changed in version 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

socket.listen([*backlog*])
Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Changed in version 3.5: The *backlog* parameter is now optional.

socket.makefile(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
Return a *file object* associated with the socket. The exact returned type depends on the arguments given to *makefile()*. These arguments are interpreted the same way as by the built-in *open()* function, except the only supported *mode* values are `'r'` (default), `'w'` and `'b'`.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by *makefile()* won't close the original socket unless all other file objects have been closed and *socket.close()* has been called on the socket object.

Note: On Windows, the file-like object created by *makefile()* cannot be used where a file object with a file descriptor is expected, such as the stream arguments of *subprocess.Popen()*.

`socket.recv(`*bufsize*[, *flags*]`)`

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

Note: For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

`socket.recvfrom(`*bufsize*[, *flags*]`)`

Receive data from the socket. The return value is a pair `(bytes, address)` where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

Changed in version 3.7: For multicast IPv6 address, first item of *address* does not contain `%scope` part anymore. In order to get full IPv6 address use *getnameinfo()*.

`socket.recvmsg(`*bufsize*[, *ancbufsize*[, *flags*]]`)`

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using *CMSG_SPACE()* or *CMSG_LEN()*, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for *recv()*.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a *bytes* object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a *bytes* object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, *sendmsg()* and *recvmsg()* can be used to pass file descriptors between processes over an *AF_UNIX* socket. When this facility is used (it is often restricted to *SOCK_STREAM* sockets), *recvmsg()* will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a *bytes* object representing the new file descriptors as a binary array of the native C `int` type. If *recvmsg()* raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, *recvmsg()* will issue

a *RuntimeWarning*, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the SCM_RIGHTS mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also *sendmsg()*.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS):
            # Append data, ignoring any truncated integers at the end.
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as *recvmsg()* would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. *bytearray* objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (*sysconf()* value SC_IOV_MAX) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for *recvmsg()*.

The return value is a 4-tuple: (nbytes, ancdata, msg_flags, address), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for *recvmsg()*.

Example:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

socket.recvfrom_into(*buffer*[, *nbytes*[, *flags*]])
: Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.recv_into(*buffer*[, *nbytes*[, *flags*]])
: Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

socket.send(*bytes*[, *flags*])
: Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the socket-howto.

  Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.sendall(*bytes*[, *flags*])
: Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()*, this method continues to send data from *bytes* 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 sent.

  Changed in version 3.5: The socket timeout is no more reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

  Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.sendto(*bytes*, *address*)
socket.sendto(*bytes*, *flags*, *address*)
: Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for *recv()* above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

  Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.sendmsg(*buffers*[, *ancdata*[, *flags*[, *address*]]])
: Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. *bytes* objects); the operating system may set a limit (*sysconf()* value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without *CMSG_SPACE()*) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for *send()*. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number

of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an *AF_UNIX* socket, on systems which support the SCM_RIGHTS mechanism. See also *recvmsg()*.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.array(
↪"i", fds))])
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendmsg_afalg**([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])

Specialized version of *sendmsg()* for *AF_ALG* socket. Set mode, IV, AEAD associated data length and flags for *AF_ALG* socket.

Availability: Linux >= 2.6.38.

New in version 3.6.

socket.**sendfile**(*file*, *offset=0*, *count=None*)

Send a file until EOF is reached by using high-performance *os.sendfile* and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If *os.sendfile* is not available (e.g. Windows) or *file* is not a regular file *send()* will be used instead. *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 updated on return or also in case of error in which case *file.tell()* can be used to figure out the number of bytes which were sent. The socket must be of *SOCK_STREAM* type. Non-blocking sockets are not supported.

New in version 3.5.

socket.**set_inheritable**(*inheritable*)

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

New in version 3.4.

socket.**setblocking**(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *settimeout()* calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

Changed in version 3.7: The method no longer applies *SOCK_NONBLOCK* flag on *socket.type*.

socket.**settimeout**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

Changed in version 3.7: The method no longer toggles *SOCK_NONBLOCK* flag on *socket.type*.

socket.**setsockopt**(*level*, *optname*, *value: int*)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in the *socket* module (`SO_*` etc.). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When value is set to `None`, optlen argument is required. It's equivalent to call setsockopt C function with optval=NULL and optlen=optlen.

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

Changed in version 3.6: setsockopt(level, optname, None, optlen: int) form added.

socket.**shutdown**(*how*)

Shut down one or both halves of the connection. If *how* is `SHUT_RD`, further receives are disallowed. If *how* is `SHUT_WR`, further sends are disallowed. If *how* is `SHUT_RDWR`, further sends and receives are disallowed.

socket.**share**(*process_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using *fromshare()*. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability: Windows.

New in version 3.3.

Note that there are no methods `read()` or `write()`; use *recv()* and *send()* without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the *socket* constructor.

socket.**family**

The socket family.

socket.**type**

The socket type.

socket.**proto**

The socket protocol.

19.2.4 Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling *setdefaulttimeout()*.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the *select* can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a *timeout* exception) or if the system returns an error.

Note: At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the *fileno()* of a socket.

Timeouts and the connect method

The *connect()* operation is also subject to the timeout setting, and in general it is recommended to call *settimeout()* before calling *connect()* or pass a timeout parameter to *create_connection()*. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the accept method

If *getdefaulttimeout()* is not *None*, sockets returned by the *accept()* method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by *accept()* is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by *accept()* is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

19.2.5 Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence *socket()*, *bind()*, *listen()*, *accept()* (possibly repeating the *accept()* to service more than one client), while a client only needs the sequence *socket()*, *connect()*. Also note that the server does not *sendall()*/*recv()* on the socket it is listening on but on the new socket returned by *accept()*.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
```

(continues on next page)

(continued from previous page)

```
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (*CAN_BCM*) the socket, you can use the *socket.send()*, and the *socket.recv()* operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

See also:

For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

19.3 `ssl` — TLS/SSL wrapper for socket objects

Source code: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

Warning: Don't use this module without reading the *Security considerations*. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the "See Also" section at the bottom.

This module provides a class, *`ssl.SSLSocket`*, which is derived from the *`socket.socket`* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, and `cipher()`,which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the *`ssl.SSLContext`* class helps manage settings and certificates, which can then be inherited by SSL sockets created through the *`SSLContext.wrap_socket()`* method.

Changed in version 3.5.3: Updated to support linking with OpenSSL 1.1.0

Changed in version 3.6: OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.

19.3.1 Functions, Constants, and Exceptions

Socket creation

Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* of an *SSLContext* instance to wrap sockets as *SSLSocket* objects. The helper functions *create_default_context()* returns a new context with secure default settings. The old *wrap_socket()* function is deprecated since it is both inefficient and has no support for server name indication (SNI) and hostname matching.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

Context creation

A convenience function helps create *SSLContext* objects for common purposes.

ssl.create_default_context(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Return a new *SSLContext* object with default settings for the given *purpose*. The settings are chosen by the *ssl* module, and usually represent a higher security level than when calling the *SSLContext* constructor directly.

cafile, *capath*, *cadata* represent optional CA certificates to trust for certificate verification, as in *SSLContext.load_verify_locations()*. If all three are *None*, this function can choose to trust the system's default CA certificates instead.

The settings are: *PROTOCOL_TLS*, *OP_NO_SSLv2*, and *OP_NO_SSLv3* with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing *SERVER_AUTH* as *purpose* sets *verify_mode* to *CERT_REQUIRED* and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses *SSLContext.load_default_certs()* to load default CA certificates.

Note: The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a *SSLContext* and apply the settings yourself.

Note: If you find that when certain older clients or servers attempt to connect with a *SSLContext* created by this function that they get an error stating "Protocol or cipher suite mismatch", it may be that they only support SSL3.0 which this function excludes using the *OP_NO_SSLv3*. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

New in version 3.4.

Changed in version 3.4.4: RC4 was dropped from the default cipher string.

Changed in version 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Exceptions

exception ssl.SSLError

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of *OSError*. The error code and message of *SSLError* instances are provided by the OpenSSL library.

Changed in version 3.3: *SSLError* used to be a subtype of *socket.error*.

library

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as `SSL`, `PEM` or `X509`. The range of possible values depends on the OpenSSL version.

New in version 3.3.

reason

A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

New in version 3.3.

exception ssl.SSLZeroReturnError
A subclass of *SSLError* raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

New in version 3.3.

exception ssl.SSLWantReadError
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

New in version 3.3.

exception ssl.SSLWantWriteError
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

New in version 3.3.

exception ssl.SSLSyscallError
A subclass of *SSLError* raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

New in version 3.3.

exception ssl.SSLEOFError
A subclass of *SSLError* raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

New in version 3.3.

exception ssl.SSLCertVerificationError
A subclass of *SSLError* raised when certificate validation has failed.

New in version 3.7.

verify_code
A numeric error number that denotes the verification error.

verify_message
A human readable string of the verification error.

exception ssl.CertificateError
An alias for *SSLCertVerificationError*.

Changed in version 3.7: The exception is now an alias for *SSLCertVerificationError*.

Random generation

ssl.RAND_bytes(*num*)
Return *num* cryptographically strong pseudo-random bytes. Raises an *SSLError* if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. *RAND_status()* can be used to check the status of the PRNG and *RAND_add()* can be used to seed the PRNG.

For almost all applications *os.urandom()* is preferable.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically generator.

New in version 3.3.

ssl.RAND_pseudo_bytes(*num*)
Return (bytes, is_cryptographic): bytes are *num* pseudo-random bytes, is_cryptographic is `True` if

the bytes generated are cryptographically strong. Raises an *SSLError* if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

For almost all applications *os.urandom()* is preferable.

New in version 3.3.

Deprecated since version 3.6: OpenSSL has deprecated *ssl.RAND_pseudo_bytes()*, use *ssl.RAND_bytes()* instead.

ssl.RAND_status()
Return `True` if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and `False` otherwise. You can use *ssl.RAND_egd()* and *ssl.RAND_add()* to increase the randomness of the pseudo-random number generator.

ssl.RAND_egd(*path*)
If you are running an entropy-gathering daemon (EGD) somewhere, and *path* is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for sources of entropy-gathering daemons.

Availability: not available with LibreSSL and OpenSSL > 1.1.0.

ssl.RAND_add(*bytes*, *entropy*)
Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

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

Certificate handling

ssl.match_hostname(*cert*, *hostname*)
Verify that *cert* (in decoded format as returned by *SSLSocket.getpeercert()*) matches the given *hostname*. The rules applied are those for checking the identity of HTTPS servers as outlined in **RFC 2818**, **RFC 5280** and **RFC 6125**. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

CertificateError is raised on failure. On success, the function returns nothing:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

New in version 3.2.

Changed in version 3.3.3: The function now follows **RFC 6125**, section 6.4.3 and does neither match multiple wildcards (e.g. `*.*.com` or `*a*.example.org`) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as `www*.xn--pthon-kva.org` are still supported, but `x*.python.org` no longer matches `xn--tda.python.org`.

Changed in version 3.5: Matching of IP addresses, when present in the subjectAltName field of the certificate, is now supported.

Changed in version 3.7: The function is no longer used to TLS connections. Hostname matching is now performed by OpenSSL.

Allow wildcard when it is the leftmost and the only character in that segment. Partial wildcards like `www*.example.com` are no longer supported.

Deprecated since version 3.7.

`ssl.cert_time_to_seconds(cert_time)`
Return the time in seconds since the Epoch, given the `cert_time` string representing the "notBefore" or "notAfter" date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here's an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (**RFC 5280**).

Changed in version 3.5: Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

`ssl.get_server_certificate(addr, ssl_version=PROTOCOL_TLS, ca_certs=None)`
Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in *SSLContext.wrap_socket()*. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

Changed in version 3.3: This function is now IPv6-compatible.

Changed in version 3.5: The default *ssl_version* is changed from *PROTOCOL_SSLv3* to *PROTOCOL_TLS* for maximum compatibility with modern servers.

`ssl.DER_cert_to_PEM_cert(DER_cert_bytes)`
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

`ssl.PEM_cert_to_DER_cert(PEM_cert_string)`
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

`ssl.get_default_verify_paths()`
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by *SSLContext.set_default_verify_paths()*. The return value is a *named tuple* `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn't exist,
- `capath` - resolved path to capath or `None` if the directory doesn't exist,
- `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,

- `openssl_capath_env` - OpenSSL's environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars `openssl_cafile_env` and `openssl_capath_env`.

New in version 3.4.

`ssl.enum_certificates`(*store_name*)

Retrieve certificates from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Example:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

New in version 3.4.

`ssl.enum_crls`(*store_name*)

Retrieve CRLs from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Availability: Windows.

New in version 3.4.

`ssl.wrap_socket`(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)

Takes an instance `sock` of *socket.socket*, and returns an instance of *ssl.SSLSocket*, a subtype of *socket.socket*, which wraps the underlying socket in an SSL context. `sock` must be a *SOCK_STREAM* socket; other socket types are unsupported.

Internally, function creates a *SSLContext* with protocol *ssl_version* and *SSLContext.options* set to *cert_reqs*. If parameters *keyfile*, *certfile*, *ca_certs* or *ciphers* are set, then the values are passed to *SSLContext.load_cert_chain()*, *SSLContext.load_verify_locations()*, and *SSLContext.set_ciphers()*.

The arguments *server_side*, *do_handshake_on_connect*, and *suppress_ragged_eofs* have the same meaning as *SSLContext.wrap_socket()*.

Deprecated since version 3.7: Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* instead of *wrap_socket()*. The top-level function is limited and creates an insecure client socket without server name indication or hostname matching.

Constants

All constants are now *enum.IntEnum* or *enum.IntFlag* collections.

New in version 3.6.

ssl.CERT_NONE
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of *Security considerations* below.

ssl.CERT_OPTIONAL
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

ssl.CERT_REQUIRED
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

class ssl.VerifyMode
enum.IntEnum collection of CERT_* constants.

New in version 3.6.

ssl.VERIFY_DEFAULT
Possible value for *SSLContext.verify_flags*. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

New in version 3.4.

ssl.VERIFY_CRL_CHECK_LEAF
Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is check but non of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded *SSLContext.load_verify_locations*, validation will fail.

New in version 3.4.

ssl.VERIFY_CRL_CHECK_CHAIN
Possible value for *SSLContext.verify_flags*. In this mode, CRLs of all certificates in the peer cert chain are checked.

New in version 3.4.

ssl.VERIFY_X509_STRICT
Possible value for *SSLContext.verify_flags* to disable workarounds for broken X.509 certificates.

New in version 3.4.

ssl.VERIFY_X509_TRUSTED_FIRST
Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

New in version 3.4.4.

class ssl.VerifyFlags
enum.IntFlag collection of VERIFY_* constants.

New in version 3.6.

ssl.PROTOCOL_TLS
Selects the highest protocol version that both the client and server support. Despite the name, this option can select both "SSL" and "TLS" protocols.

New in version 3.6.

ssl.PROTOCOL_TLS_CLIENT
Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support client-side *SSLSocket* connections. The protocol enables *CERT_REQUIRED* and *check_hostname* by default.

New in version 3.6.

ssl.PROTOCOL_TLS_SERVER
Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support server-side *SSLSocket* connections.

New in version 3.6.

ssl.PROTOCOL_SSLv23
Alias for data:*PROTOCOL_TLS*.

Deprecated since version 3.6: Use *PROTOCOL_TLS* instead.

ssl.PROTOCOL_SSLv2
Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `OPENSSL_NO_SSL2` flag.

> **Warning:** SSL version 2 is insecure. Its use is highly discouraged.

Deprecated since version 3.6: OpenSSL has removed support for SSLv2.

ssl.PROTOCOL_SSLv3
Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the `OPENSSL_NO_SSLv3` flag.

> **Warning:** SSL version 3 is insecure. Its use is highly discouraged.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1
Selects TLS version 1.0 as the channel encryption protocol.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1_1
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

New in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.PROTOCOL_TLSv1_2
Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

New in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.OP_ALL
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's SSL_OP_ALL constant.

New in version 3.2.

ssl.OP_NO_SSLv2
Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

New in version 3.2.

Deprecated since version 3.6: SSLv2 is deprecated

ssl.OP_NO_SSLv3
Prevents an SSLv3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv3 as the protocol version.

New in version 3.2.

Deprecated since version 3.6: SSLv3 is deprecated

ssl.OP_NO_TLSv1
Prevents a TLSv1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1 as the protocol version.

New in version 3.2.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0, use the new *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.OP_NO_TLSv1_1
Prevents a TLSv1.1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

New in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_2
Prevents a TLSv1.2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

New in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_3
: Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

  New in version 3.7.

  Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15, 3.6.3 and 3.7.0 for backwards compatibility with OpenSSL 1.0.2.

ssl.OP_NO_RENEGOTIATION
: Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

  This option is only available with OpenSSL 1.1.0h and later.

  New in version 3.7.

ssl.OP_CIPHER_SERVER_PREFERENCE
: Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

  New in version 3.3.

ssl.OP_SINGLE_DH_USE
: Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

  New in version 3.3.

ssl.OP_SINGLE_ECDH_USE
: Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

  New in version 3.3.

ssl.OP_ENABLE_MIDDLEBOX_COMPAT
: Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

  This option is only available with OpenSSL 1.1.1 and later.

  New in version 3.8.

ssl.OP_NO_COMPRESSION
: Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

  This option is only available with OpenSSL 1.0.0 and later.

  New in version 3.3.

class ssl.Options
: *enum.IntFlag* collection of OP_* constants.

ssl.OP_NO_TICKET
: Prevent client side from requesting a session ticket.

  New in version 3.6.

ssl.HAS_ALPN
: Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301**.

  New in version 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**
Whether the OpenSSL library has built-in support not checking subject common name and *SSLContext.hostname_checks_common_name* is writeable.

New in version 3.7.

ssl.**HAS_ECDH**
Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

New in version 3.3.

ssl.**HAS_SNI**
Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 6066**).

New in version 3.2.

ssl.**HAS_NPN**
Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the Application Layer Protocol Negotiation. When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

New in version 3.3.

ssl.**HAS_SSLv2**
Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

New in version 3.7.

ssl.**HAS_SSLv3**
Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

New in version 3.7.

ssl.**HAS_TLSv1**
Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

New in version 3.7.

ssl.**HAS_TLSv1_1**
Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

New in version 3.7.

ssl.**HAS_TLSv1_2**
Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

New in version 3.7.

ssl.**HAS_TLSv1_3**
Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

New in version 3.7.

ssl.**CHANNEL_BINDING_TYPES**
List of supported TLS channel binding types. Strings in this list can be used as arguments to *SSLSocket.get_channel_binding()*.

New in version 3.3.

ssl.**OPENSSL_VERSION**
The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

New in version 3.2.

ssl.OPENSSL_VERSION_INFO
A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

New in version 3.2.

ssl.OPENSSL_VERSION_NUMBER
The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

New in version 3.2.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl.ALERT_DESCRIPTION_INTERNAL_ERROR
ALERT_DESCRIPTION_*
Alert Descriptions from **RFC 5246** and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in *SSLContext.set_servername_callback()*.

New in version 3.4.

class ssl.AlertDescription
enum.IntEnum collection of ALERT_DESCRIPTION_* constants.

New in version 3.6.

Purpose.SERVER_AUTH
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

New in version 3.4.

Purpose.CLIENT_AUTH
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

New in version 3.4.

class ssl.SSLErrorNumber
enum.IntEnum collection of SSL_ERROR_* constants.

New in version 3.6.

class ssl.TLSVersion
enum.IntEnum collection of SSL and TLS versions for *SSLContext.maximum_version* and *SSLContext.minimum_version*.

New in version 3.7.

TLSVersion.MINIMUM_SUPPORTED

`TLSVersion.MAXIMUM_SUPPORTED`
The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

`TLSVersion.SSLv3`

`TLSVersion.TLSv1`

`TLSVersion.TLSv1_1`

`TLSVersion.TLSv1_2`

`TLSVersion.TLSv1_3`
SSL 3.0 to TLS 1.3.

19.3.2 SSL Sockets

`class ssl.SSLSocket(`*socket.socket*`)`
SSL sockets provide the following methods of *Socket Objects*:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)
- *shutdown()*

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

Instances of *SSLSocket* must be created using the *SSLContext.wrap_socket()* method.

Changed in version 3.5: The `sendfile()` method was added.

Changed in version 3.5: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now to maximum total duration of the shutdown.

Deprecated since version 3.6: It is deprecated to create a *SSLSocket* instance directly, use *SSLContext.wrap_socket()* to wrap a socket.

Changed in version 3.7: *SSLSocket* instances must to created with *wrap_socket()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSL sockets also have the following additional methods and attributes:

SSLSocket.**read**(*len=1024*, *buffer=None*)

Read up to *len* bytes of data from the SSL socket and return the result as a `bytes` instance. If *buffer* is specified, then read into the buffer instead, and return the number of bytes read.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the read would block.

As at any time a re-negotiation is possible, a call to *read()* can also cause write operations.

Changed in version 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to read up to *len* bytes.

Deprecated since version 3.6: Use `recv()` instead of *read()*.

SSLSocket.**write**(*buf*)

Write *buf* to the SSL socket and return the number of bytes written. The *buf* argument must be an object supporting the buffer interface.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the write would block.

As at any time a re-negotiation is possible, a call to *write()* can also cause read operations.

Changed in version 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to write *buf*.

Deprecated since version 3.6: Use `send()` instead of *write()*.

Note: The *read()* and *write()* methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and *SSLSocket.unwrap()* was not called.

Normally you should use the socket API methods like *recv()* and *send()* instead of these methods.

SSLSocket.**do_handshake**()

Perform the SSL setup handshake.

Changed in version 3.4: The handshake method also performs *match_hostname()* when the *check_hostname* attribute of the socket's *context* is true.

Changed in version 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration of the handshake.

Changed in version 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function *match_hostname()* is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is send to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn't been done yet, raise *ValueError*.

If the `binary_form` parameter is *False*, and a certificate was received from the peer, this method returns a *dict* instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

Note: To validate a certificate for a particular service, you can use the *match_hostname()* function.

If the `binary_form` parameter is *True*, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or *None* if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore *getpeercert()* will return *None* if you used *CERT_NONE* (rather than *CERT_OPTIONAL* or *CERT_REQUIRED*).

Changed in version 3.2: The returned dictionary includes additional items such as `issuer` and `notBefore`.

Changed in version 3.4: *ValueError* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

`SSLSocket.cipher()`
Returns 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. If no connection has been established, returns `None`.

`SSLSocket.shared_ciphers()`
Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns `None` if no connection has been established or the socket is a client socket.

New in version 3.5.

`SSLSocket.compression()`
Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

New in version 3.3.

SSLSocket.**get_channel_binding**(*cb_type="tls-unique"*)
Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

New in version 3.3.

SSLSocket.**selected_alpn_protocol**()
Return the protocol that was selected during the TLS handshake. If *SSLContext.set_alpn_protocols()* was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

New in version 3.5.

SSLSocket.**selected_npn_protocol**()
Return the higher-level protocol that was selected during the TLS/SSL handshake. If *SSLContext.set_npn_protocols()* was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

New in version 3.3.

SSLSocket.**unwrap**()
Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket.**verify_client_post_handshake**()
Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see *SSLContext.post_handshake_auth*.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an *SSLError* is raised.

New in version 3.7.1.

Note: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises *NotImplementedError*.

SSLSocket.**version**()
Return the actual SSL protocol version negotiated by the connection as a string, or `None` is no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

New in version 3.5.

SSLSocket.**pending**()
Returns the number of already decrypted bytes available for read, pending on the connection.

SSLSocket.**context**
The *SSLContext* object this SSL socket is tied to. If the SSL socket was created using the deprecated

wrap_socket() function (rather than *SSLContext.wrap_socket()*), this is a custom context object created for this SSL socket.

New in version 3.2.

SSLSocket.server_side
A boolean which is `True` for server-side sockets and `False` for client-side sockets.

New in version 3.2.

SSLSocket.server_hostname
Hostname of the server: *str* type, or `None` for server-side socket or if the hostname was not specified in the constructor.

New in version 3.2.

Changed in version 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

SSLSocket.session
The *SSLSession* for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before *do_handshake()* has been called to reuse a session.

New in version 3.6.

SSLSocket.session_reused
New in version 3.6.

19.3.3 SSL Contexts

New in version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.SSLContext(*protocol=PROTOCOL_TLS*)
Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	yes	no	no[1]	no	no	no
SSLv3	no	yes	no[2]	no	no	no
TLS (SSLv23)[3]	no[1]	no[2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

[3] TLS 1.3 protocol will be available with *PROTOCOL_TLS* in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.
[1] *SSLContext* disables SSLv2 with *OP_NO_SSLv2* by default.
[2] *SSLContext* disables SSLv3 with *OP_NO_SSLv3* by default.

See also:

create_default_context() lets the *ssl* module choose security settings for a given purpose.

Changed in version 3.6: The context is created with secure default values. The options *OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (except for *PROTOCOL_SSLv2*), and *OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*) are set by default. The initial cipher suite list contains only `HIGH` ciphers, no `NULL` ciphers and no `MD5` ciphers (except for *PROTOCOL_SSLv2*).

SSLContext objects have the following methods and attributes:

SSLContext.cert_store_stats()

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

New in version 3.4.

SSLContext.load_cert_chain(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn't match with the certificate.

Changed in version 3.3: New optional argument *password*.

SSLContext.load_default_certs(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On other systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings *Purpose.SERVER_AUTH* loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

New in version 3.4.

SSLContext.load_verify_locations(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Changed in version 3.4: New optional argument *cadata*

`SSLContext.get_ca_certs(`*binary_form=False*`)`
Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

Note: Certificates in a capath directory aren't loaded unless they have been used at least once.

New in version 3.4.

`SSLContext.get_ciphers()`
Get a list of enabled ciphers. The list is in order of cipher priority. See *SSLContext.set_ciphers()*.

Example:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

On OpenSSL 1.1 and newer the cipher dict contains additional fields:

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
```

(continues on next page)

(continued from previous page)

```
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Availability: OpenSSL 1.0.2+.

New in version 3.6.

SSLContext.set_default_verify_paths()

Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(*ciphers*)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

Note: when connected, the *SSLSocket.cipher()* method of SSL sockets will give the currently selected cipher.

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with *set_ciphers()*.

SSLContext.set_alpn_protocols(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the *SSLSocket.selected_alpn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_ALPN* is False.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise *SSLError* when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, *SSLSocket.selected_alpn_protocol()* returns None.

New in version 3.5.

SSLContext.set_npn_protocols(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol

Negotiation. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_NPN* is False.

New in version 3.3.

SSLContext.sni_callback

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the *ssl.SSLSocket*'s *SSLSocket.context* attribute to a new object of type *SSLContext* representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. *SSLSocket.getpeercert()*, *SSLSocket.getpeercert()*, *SSLSocket.cipher()* and `SSLSocket.compress()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

This method will raise *NotImplementedError* if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

New in version 3.7.

SSLContext.set_servername_callback(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use *sni_callback* instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is an decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

New in version 3.4.

SSLContext.load_dh_params(*dhfile*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_DH_USE* option to further improve security.

New in version 3.3.

SSLContext.set_ecdh_curve(*curve_name*)
Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_ECDH_USE* option to further improve security.

This method is not available if *HAS_ECDH* is `False`.

New in version 3.3.

See also:

SSL/TLS & Perfect Forward Secrecy Vincent Bernat.

SSLContext.wrap_socket(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)
Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class* (default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise *SSLError*.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the *SSLSocket.do_handshake()* method. Calling *SSLSocket.do_handshake()* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *False*, it will raise the exceptions back to the caller.

session, see *session*.

Changed in version 3.5: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

Changed in version 3.6: *session* argument was added.

Changed in version 3.7: The method returns on instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

SSLContext.sslsocket_class
The return type of `SSLContext.wrap_sockets()`, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

New in version 3.7.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of attr:*SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

Changed in version 3.6: *session* argument was added.

Changed in version 3.7: The method returns on instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

SSLContext.**sslobject_class**

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

New in version 3.7.

SSLContext.**session_stats**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

Whether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled.

Example:

```
import socket, ssl

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

New in version 3.4.

Changed in version 3.7: *verify_mode* is now automatically changed to *CERT_REQUIRED* when hostname checking is enabled and *verify_mode* is *CERT_NONE*. Previously the same operation would have failed with a *ValueError*.

Note: This features requires OpenSSL 0.9.8f or newer.

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to

TLSVersion.MAXIMUM_SUPPORTED. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

The attributes *maximum_version*, *minimum_version* and *SSLContext.options* all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with *OP_NO_TLSv1_2* in *options* and *maximum_version* set to *TLSVersion.TLSv1_2* will not be able to establish a TLS 1.2 connection.

Note: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

SSLContext.minimum_version

Like *SSLContext.maximum_version* except it is the lowest supported version or *TLSVersion.MINIMUM_SUPPORTED*.

Note: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

SSLContext.options

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

Note: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a *ValueError*.

Changed in version 3.6: *SSLContext.options* returns *Options* flags:

```
>>> ssl.create_default_context().options  # doctest: +SKIP
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.post_handshake_auth

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

New in version 3.7.1.

Note: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is None and can't be modified

SSLContext.protocol

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.hostname_checks_common_name

Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

New in version 3.7.

Note: Only writeable with OpenSSL 1.1.0 or higher.

`SSLContext.verify_flags`
The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

New in version 3.4.

Changed in version 3.6: *SSLContext.verify_flags* returns *VerifyFlags* flags:

```
>>> ssl.create_default_context().verify_flags  # doctest: +SKIP
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

`SSLContext.verify_mode`
Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

Changed in version 3.6: *SSLContext.verify_mode* returns *VerifyMode* enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

19.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling *SSLContext.load_default_certs()*, this is done automatically with *create_default_context()*.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to *SSLContext.load_cert_chain()* and *wrap_socket()* needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
```

(continues on next page)

(continued from previous page)

```
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

19.3.5 Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext()
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

When you use the context to connect to a server, *CERT_REQUIRED* validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',
↪),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro,'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
```

(continues on next page)

(continued from previous page)

```
                   ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of *Security considerations* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *SSLContext.wrap_socket()* method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

19.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most *SSLSocket* methods will raise either *SSLWantWriteError* or *SSLWantReadError* instead of *BlockingIOError* if an I/O operation would block. *SSLWantReadError* will be raised if a read operation on the underlying socket is necessary, and *SSLWantWriteError* for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

  Changed in version 3.5: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising *SSLWantWriteError* or *SSLWantReadError*.

- Calling *select()* tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to *select()*.

- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without *select()* being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a *select()* call if still necessary.

  (of course, similar provisions apply when using other primitives such as *poll()*, or those in the *selectors* module)

- The SSL handshake itself will be non-blocking: the *SSLSocket.do_handshake()* method has to be retried until it returns successfully. Here is a synopsis using *select()* to wait for the socket's readiness:

  ```
  while True:
      try:
          sock.do_handshake()
          break
      except ssl.SSLWantReadError:
          select.select([sock], [], [])
      except ssl.SSLWantWriteError:
          select.select([], [sock], [])
  ```

See also:

The *asyncio* module supports *non-blocking SSL sockets* and provides a higher level API. It polls for events using the *selectors* module and handles *SSLWantWriteError*, *SSLWantReadError* and *BlockingIOError* exceptions. It runs the SSL handshake asynchronously as well.

19.3.7 Memory BIO Support

New in version 3.5.

Ever since the SSL module was introduced in Python 2.6, the *SSLSocket* class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by *socket.socket*, from which *SSLSocket* also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the "select/poll on a file descriptor" (readiness based) model that is assumed by *socket.socket* and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of *SSLSocket* called *SSLObject* is provided.

class ssl.SSLObject

A reduced-scope variant of *SSLSocket* representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate "BIO" objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An *SSLObject* instance must be created using the *wrap_bio()* method. This method will create the *SSLObject* instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*

- *do_handshake()*
- *unwrap()*
- *get_channel_binding()*

When compared to *SSLSocket*, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying *MemoryBIO* buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call *do_handshake()* to start the handshake.
- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the *SSLEOFError* exception.
- The method *unwrap()* call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to *SSLContext.set_servername_callback()* will get an *SSLObject* instance instead of a *SSLSocket* instance as its first parameter.

Some notes related to the use of *SSLObject*:

- All IO on an *SSLObject* is *non-blocking*. This means that for example *read()* will raise an *SSLWantReadError* if it needs more data than the incoming BIO has available.
- There is no module-level `wrap_bio()` call like there is for *wrap_socket()*. An *SSLObject* is always created via an *SSLContext*.

Changed in version 3.7: *SSLObject* instances must to created with *wrap_bio()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class *MemoryBIO* provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class ssl.MemoryBIO

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call *write()*. The attribute *eof* will become true after all data currently in the buffer has been read.

19.3.8 SSL session

New in version 3.6.

class ssl.SSLSession

Session object used by *session*.

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the *create_default_context()* function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the *smtplib.SMTP* class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the *match_hostname()* function can be used. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

Changed in version 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses *match_hostname()*.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

The SSL context created above will only allow TLSv1.2 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()*, *RAND_bytes()* or *RAND_pseudo_bytes()* is sufficient.

19.3.10 TLS 1.3

New in version 3.7.

Python has provisional and experimental support for TLS 1.3 with OpenSSL 1.1.1. The new protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

19.3.11 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods *SSLContext.set_npn_protocols()* and *SSLSocket.selected_npn_protocol()* are not available.
- *SSLContext.set_default_verify_paths()* ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although *get_default_verify_paths()* still reports them.

See also:

Class *socket.socket* Documentation of underlying *socket* class

SSL/TLS Strong Encryption: An Introduction Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management Steve Kent

RFC 4086: Randomness Requirements for Security Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Pro D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport IETF

Mozilla's Server Side TLS recommendations Mozilla

19.4 `select` — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

Note: The *selectors* module allows high-level and efficient I/O multiplexing, built upon the *select* module primitives. Users are encouraged to use the *selectors* module instead, unless they want precise control over the OS-level primitives used.

The module defines the following:

`exception select.error`
A deprecated alias of *OSError*.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of *OSError*.

`select.devpoll()`
(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

New in version 3.3.

Changed in version 3.4: The new file descriptor is now non-inheritable.

`select.epoll`(*sizehint=-1*, *flags=0*)

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or *-1* to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

Changed in version 3.3: Added the *flags* parameter.

Changed in version 3.4: Support for the `with` statement was added. The new file descriptor is now non-inheritable.

Deprecated since version 3.4: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use *`os.set_inheritable()`* to make the file descriptor inheritable.

`select.poll`()

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section *Polling Objects* below for the methods supported by polling objects.

`select.kqueue`()

(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

Changed in version 3.4: The new file descriptor is now non-inheritable.

`select.kevent`(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

`select.select`(*rlist*, *wlist*, *xlist*[, *timeout*])

This is a straightforward interface to the Unix `select()` system call. The first three arguments are sequences of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named *`fileno()`* returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python *file objects* (e.g. `sys.stdin`, or objects returned by *`open()`* or *`os.popen()`*), socket objects returned by *`socket.socket()`*. You may also define a *wrapper* class yourself, as long as it has an appropriate *`fileno()`* method (that really returns a file descriptor, not just a random integer).

Note: File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *`InterruptedError`*.

`select.PIPE_BUF`
: The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by *`select()`*, *`poll()`* or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

  This value is guaranteed by POSIX to be at least 512.

  Availability: Unix

  New in version 3.2.

19.4.1 `/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is O(highest file descriptor) and `poll()` is O(number of file descriptors), `/dev/poll` is O(active file descriptors).

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.close()`
: Close the file descriptor of the polling object.

  New in version 3.4.

`devpoll.closed`
: `True` if the polling object is closed.

  New in version 3.4.

`devpoll.fileno()`
: Return the file descriptor number of the polling object.

  New in version 3.4.

`devpoll.register(`*fd*[, *eventmask*]`)`
: Register a file descriptor with the polling object. Future calls to the *`poll()`* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *`fileno()`* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

> **Warning:** Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

`devpoll.modify(`*fd*[, *eventmask*]`)`
This method does an *`unregister()`* followed by a *`register()`*. It is (a bit) more efficient that doing the same explicitly.

`devpoll.unregister(`*fd*`)`
Remove a file descriptor being tracked by a polling object. Just like the *`register()`* method, *fd* can be an integer or an object with a *`fileno()`* method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

`devpoll.poll(`[*timeout*]`)`
Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *`None`*, the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *`InterruptedError`*.

19.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constant	Meaning
`EPOLLIN`	Available for read
`EPOLLOUT`	Available for write
`EPOLLPRI`	Urgent data for read
`EPOLLERR`	Error condition happened on the assoc. fd
`EPOLLHUP`	Hang up happened on the assoc. fd
`EPOLLET`	Set Edge Trigger behavior, the default is Level Trigger behavior
`EPOLLONESHOT`	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
`EPOLLEXCLUSIVE`	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
`EPOLLRDHUP`	Stream socket peer closed connection or shut down writing half of connection.
`EPOLLRDNORM`	Equivalent to `EPOLLIN`
`EPOLLRDBAND`	Priority data band can be read.
`EPOLLWRNORM`	Equivalent to `EPOLLOUT`
`EPOLLWRBAND`	Priority data may be written.
`EPOLLMSG`	Ignored.

New in version 3.6: `EPOLLEXCLUSIVE` was added. It's only supported by Linux Kernel 4.5 or later.

`epoll.close()`
Close the control file descriptor of the epoll object.

`epoll.closed`
`True` if the epoll object is closed.

`epoll.fileno()`
Return the file descriptor number of the control fd.

`epoll.fromfd(`*fd*`)`
Create an epoll object from a given file descriptor.

`epoll.register(`*fd*[, *eventmask*]`)`
Register a fd descriptor with the epoll object.

`epoll.modify(`*fd*, *eventmask*`)`
Modify a registered file descriptor.

`epoll.unregister(`*fd*`)`
Remove a registered file descriptor from the epoll object.

`epoll.poll(`*timeout=-1*, *maxevents=-1*`)`
Wait for events. timeout in seconds (float)

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is O(highest file descriptor), while `poll()` is O(number of file descriptors).

`poll.register(`*fd*[, *eventmask*]`)`
Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constant	Meaning
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output: writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.modify(`*fd*, *eventmask*`)`
Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *OSError* exception with errno `ENOENT` to be raised.

`poll.unregister(`*fd*`)`
Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a *KeyError* exception to be raised.

`poll.poll(`[*timeout*]`)`
Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *None*, the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.4 Kqueue Objects

`kqueue.close()`
Close the control file descriptor of the kqueue object.

`kqueue.closed`
`True` if the kqueue object is closed.

`kqueue.fileno()`
Return the file descriptor number of the control fd.

`kqueue.fromfd(`*fd*`)`
Create a kqueue object from a given file descriptor.

`kqueue.control(`*changelist*, *max_events*[, *timeout=None*]`)` → eventlist
Low level interface to kevent

- changelist must be an iterable of kevent object or `None`
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible)

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.5 Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

`kevent.ident`
Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *fileno()* method. kevent stores the integer internally.

`kevent.filter`
Name of the kernel filter.

Constant	Meaning
KQ_FILTER_READ	Takes a descriptor and returns whenever there is data available to read
KQ_FILTER_WRITE	Takes a descriptor and returns whenever there is data available to write
KQ_FILTER_AIO	AIO requests
KQ_FILTER_VNODE	Returns when one or more of the requested events watched in *fflag* occurs
KQ_FILTER_PROC	Watch for events on a process id
KQ_FILTER_NETDEV	Watch for events on a network device [not available on Mac OS X]
KQ_FILTER_SIGNAL	Returns whenever the watched signal is delivered to the process
KQ_FILTER_TIMER	Establishes an arbitrary timer

`kevent.flags`
Filter action.

Constant	Meaning
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

`kevent.fflags`
Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

Constant	Meaning
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags:

Constant	Meaning
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

Constant	Meaning
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on Mac OS X):

Constant	Meaning
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.data
Filter specific data.

kevent.udata
User defined value.

19.5 selectors — High-level I/O multiplexing

New in version 3.4.

Source code: Lib/selectors.py

19.5.1 Introduction

This module allows high-level and efficient I/O multiplexing, built upon the *select* module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a *BaseSelector* abstract base class, along with several concrete implementations (*KqueueSelector*, *EpollSelector*...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, "file object" refers to any object with a fileno() method, or a raw file descriptor. See *file object*.

DefaultSelector is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

Note: The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

See also:

select Low-level I/O multiplexing module.

19.5.2 Classes

Classes hierarchy:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

Constant	Meaning
`EVENT_READ`	Available for read
`EVENT_WRITE`	Available for write

class selectors.**SelectorKey**

A *SelectorKey* is a *namedtuple* used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several *BaseSelector* methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.**BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector*, *KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the *context manager* protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated *SelectorKey* instance, or raises a *KeyError* if *fileobj* is not registered. It will raise *ValueError* if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()`

method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)
Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)()` followed by `BaseSelector.register(fileobj, events, data)()`, except that it can be implemented more efficiently.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is not registered.

abstractmethod select(*timeout=None*)
Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the *SelectorKey* instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

Note: This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

Changed in version 3.5: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** for the rationale), instead of returning an empty list of events before the timeout.

close()
Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)
Return the key associated with a registered file object.

This returns the *SelectorKey* instance associated to this file object, or raises *KeyError* if the file object is not registered.

abstractmethod get_map()
Return a mapping of file objects to selector keys.

This returns a *Mapping* instance mapping registered file objects to their associated *SelectorKey* instance.

class selectors.DefaultSelector
The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.SelectSelector
select.select()-based selector.

class selectors.PollSelector
select.poll()-based selector.

class selectors.EpollSelector
select.epoll()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.epoll()* object.

class selectors.DevpollSelector
select.devpoll()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.devpoll()* object.

New in version 3.5.

class selectors.KqueueSelector
select.kqueue()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.kqueue()* object.

19.5.3 Examples

Here is a simple echo server implementation:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 asyncore — Asynchronous socket handler

Source code: Lib/asyncore.py

Deprecated since version 3.6: Please use *asyncio* instead.

Note: This module exists for backwards compatibility only. For new code we recommend using *asyncio*.

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The *asyncore* module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion *asynchat* module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class *asyncore.dispatcher* and *asynchat.async_chat*. Creating the channels adds them to a global map, used by the *loop()* function if you do not provide it with your own *map*.

Once the initial channel(s) is(are) created, calling the *loop()* function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

`asyncore.loop`([*timeout*[, *use_poll*[, *map*[, *count*]]]])

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The *count* parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The *timeout* argument sets the timeout parameter for the appropriate *select()* or *poll()* call, measured in seconds; the default is 30 seconds. The *use_poll* parameter, if true, indicates that *poll()* should be used in preference to *select()* (the default is `False`).

The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used. Channels (instances of *asyncore.dispatcher*, *asynchat.async_chat* and subclasses thereof) can freely be mixed in the map.

`class asyncore.dispatcher`

The *dispatcher* class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Event	Description
`handle_connect()`	Implied by the first read or write event
`handle_close()`	Implied by a read event with no data available
`handle_accepted()`	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's *readable()* and *writable()* methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

handle_read()
Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

handle_write()
Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()
Called when the active opener's socket actually makes a connection. Might send a "welcome" banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()
Called when the socket is closed.

handle_error()
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

handle_accept()
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a *connect()* call for the local endpoint. Deprecated in version 3.2; use *handle_accepted()* instead.

Deprecated since version 3.2.

handle_accepted(*sock*, *addr*)
Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a *connect()* call for the local endpoint. *sock* is a *new* socket object usable to send and receive data on the connection, and *addr* is the address bound to the socket on the other end of the connection.

New in version 3.2.

readable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

writable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the *socket* documentation for information on creating sockets.

Changed in version 3.3: *family* and *type* arguments can be omitted.

connect(*address*)
As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

send(*data*)
Send *data* to the remote end-point of the socket.

recv(*buffer_size*)
Read at most *buffer_size* bytes from the socket's remote end-point. An empty bytes object implies that the channel has been closed from the other end.

Note that *recv()* may raise *BlockingIOError* , even though *select.select()* or *select.poll()* has reported the socket ready for reading.

listen(*backlog*)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind(*address*)
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — refer to the *socket* documentation for more information.) To mark the socket as re-usable (setting the `SO_REUSEADDR` option), call the *dispatcher* object's `set_reuse_addr()` method.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either `None` or 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. When `None` is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close()
Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class asyncore.dispatcher_with_send
A *dispatcher* subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use *asynchat.async_chat*.

class asyncore.file_dispatcher
A file_dispatcher takes a file descriptor or *file object* along with an optional map argument and wraps it for use with the `poll()` or `loop()` functions. If provided a file object or anything with a `fileno()` method, that method will be called and passed to the *file_wrapper* constructor.

Availability: Unix.

class asyncore.file_wrapper
A file_wrapper takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class.

Availability: Unix.

19.6.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the *dispatcher* class to implement its socket handling:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

19.6.2 asyncore Example basic echo server

Here is a basic echo server that uses the *dispatcher* class to accept connections and dispatches the incoming connections to a handler:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
```

(continues on next page)

(continued from previous page)

```
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

19.7 asynchat — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

Deprecated since version 3.6: Please use *asyncio* instead.

Note: This module exists for backwards compatibility only. For new code we recommend using *asyncio*.

This module builds on the *asyncore* infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. *asynchat* defines the abstract class *async_chat* that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as *asyncore*, and the two types of channel, *asyncore.dispatcher* and *asynchat.async_chat*, can freely be mixed in the channel map. Typically an *asyncore.dispatcher* server channel generates new *asynchat.async_chat* channel objects as it receives incoming connection requests.

class asynchat.async_chat

This class is an abstract subclass of *asyncore.dispatcher*. To make practical use of the code you must subclass *async_chat*, providing meaningful *collect_incoming_data()* and *found_terminator()* methods. The *asyncore.dispatcher* methods can be used, although not all make sense in a message/response context.

Like *asyncore.dispatcher*, *async_chat* defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the *async_chat* object's methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

ac_in_buffer_size

The asynchronous input buffer size (default `4096`).

ac_out_buffer_size

The asynchronous output buffer size (default `4096`).

Unlike *asyncore.dispatcher*, *async_chat* allows you to define a FIFO queue of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (*i.e.* that it contains no more data) by having its `more()` method return the empty bytes object. At this point the *async_chat* object removes the producer from the queue and starts using the next producer, if any. When the producer queue is empty the `handle_write()`

method does nothing. You use the channel object's *set_terminator()* method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning *async_chat* subclass your input methods *collect_incoming_data()* and *found_terminator()* must handle the data that the channel receives asynchronously. The methods are described below.

async_chat.close_when_done()
Pushes a `None` on to the producer queue. When this producer is popped off the queue it causes the channel to be closed.

async_chat.collect_incoming_data(*data*)
Called with *data* holding an arbitrary amount of received data. The default method, which must be overridden, raises a *NotImplementedError* exception.

async_chat.discard_buffers()
In emergencies this method will discard any data held in the input and/or output buffers and the producer queue.

async_chat.found_terminator()
Called when the incoming data stream matches the termination condition set by *set_terminator()*. The default method, which must be overridden, raises a *NotImplementedError* exception. The buffered input data should be available via an instance attribute.

async_chat.get_terminator()
Returns the current terminator for the channel.

async_chat.push(*data*)
Pushes data on to the channel's queue to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat.push_with_producer(*producer*)
Takes a producer object and adds it to the producer queue associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

async_chat.set_terminator(*term*)
Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	Description
string	Will call *found_terminator()* when the string is found in the input stream
integer	Will call *found_terminator()* when the indicated number of characters have been received
`None`	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after *found_terminator()* is called.

19.7.1 asynchat Example

The following partial example shows how HTTP requests can be read with *async_chat*. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

19.8 `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

19.8.1 General rules

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a *KeyboardInterrupt* exception.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point(for example at the next *bytecode* instruction). This has consequences:

- It makes little sense to catch synchronous errors like SIGFPE or SIGSEGV that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the *faulthandler* module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the *threading* module instead.

Besides, only the main thread is allowed to set a new signal handler.

19.8.2 Module contents

Changed in version 3.5: signal (SIG*), handler (*SIG_DFL*, *SIG_IGN*) and sigmask (*SIG_BLOCK*, *SIG_UNBLOCK*, *SIG_SETMASK*) related constants listed below were turned into *enums*. *getsignal()*, *pthread_sigmask()*, *sigpending()* and *sigwait()* functions return human-readable *enums*.

The variables defined in the *signal* module are:

signal.SIG_DFL
: This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCHLD is to simply ignore it.

signal.SIG_IGN
: This is another standard signal handler, which will simply ignore the given signal.

SIG*
: All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for 'signal()' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.CTRL_C_EVENT
The signal corresponding to the Ctrl+C keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

New in version 3.2.

signal.CTRL_BREAK_EVENT
The signal corresponding to the Ctrl+Break keystroke event. This signal can only be used with *os.kill()*.

Availability: Windows.

New in version 3.2.

signal.NSIG
One more than the number of the highest signal number.

signal.ITIMER_REAL
Decrements interval timer in real time, and delivers SIGALRM upon expiration.

signal.ITIMER_VIRTUAL
Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.ITIMER_PROF
Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.SIG_BLOCK
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be blocked.

New in version 3.3.

signal.SIG_UNBLOCK
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be unblocked.

New in version 3.3.

signal.SIG_SETMASK
A possible value for the *how* parameter to *pthread_sigmask()* indicating that the signal mask is to be replaced.

New in version 3.3.

The *signal* module defines one exception:

exception signal.ItimerError
Raised to signal an error from the underlying *setitimer()* or *getitimer()* implementation. Expect this error if an invalid interval timer or a negative time is passed to *setitimer()*. This error is a subtype of *OSError*.

New in version 3.3: This error used to be a subtype of *IOError*, which is now an alias of *OSError*.

The *signal* module defines the following functions:

signal.alarm(*time*)
If *time* is non-zero, this function requests that a SIGALRM signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page *alarm(2)*.)

Availability: Unix.

signal.getsignal(*signalnum*)
Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values *signal.SIG_IGN*, *signal.SIG_DFL* or *None*. Here, *signal.SIG_IGN* means that the signal was previously ignored, *signal.SIG_DFL* means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.pause()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page *signal(2)*.)

See also *sigwait()*, *sigwaitinfo()*, *sigtimedwait()* and *sigpending()*.

signal.pthread_kill(*thread_id*, *signalnum*)
Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Availability: Unix (see the man page *pthread_kill(3)* for further information).

See also *os.kill()*.

New in version 3.3.

signal.pthread_sigmask(*how*, *mask*)
Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- *SIG_BLOCK*: The set of blocked signals is the union of the current set and the *mask* argument.
- *SIG_UNBLOCK*: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- *SIG_SETMASK*: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {signal.SIGINT, signal.SIGTERM}). Use range(1, signal.NSIG) for a full mask including all signals.

For example, signal.pthread_sigmask(signal.SIG_BLOCK, []) reads the signal mask of the calling thread.

Availability: Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

See also *pause()*, *sigpending()* and *sigwait()*.

New in version 3.3.

signal.setitimer(*which*, *seconds*, *interval=0.0*)
Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver SIGALRM, *signal.ITIMER_VIRTUAL* sends SIGVTALRM, and *signal.ITIMER_PROF* will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*.

Availability: Unix.

signal.**getitimer**(*which*)
Returns current value of a given interval timer specified by *which*.

Availability: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)
Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Changed in version 3.5: On Windows, the function now also supports socket handles.

Changed in version 3.7: Added `warn_on_full_buffer` parameter.

signal.**siginterrupt**(*signalnum*, *flag*)
Change system call restart behaviour: if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability: Unix (see the man page *siginterrupt(3)* for further information).

Note that installing a signal handler with *signal()* will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

signal.**signal**(*signalnum*, *handler*)
Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values *signal.SIG_IGN* or *signal.SIG_DFL*. The previous signal handler will be returned (see the description of *getsignal()* above). (See the Unix man page *signal(2)*.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the *inspect* module).

On Windows, *signal()* can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A *ValueError* will be raised in any other case. Note that not all systems define the same set of signal names; an *AttributeError* will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.sigpending()`
Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix (see the man page *sigpending(2)* for further information).

See also *pause()*, *pthread_sigmask()* and *sigwait()*.

New in version 3.3.

`signal.sigwait(`*sigset*`)`
Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix (see the man page *sigwait(3)* for further information).

See also *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* and *sigtimedwait()*.

New in version 3.3.

`signal.sigwaitinfo(`*sigset*`)`
Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix (see the man page *sigwaitinfo(2)* for further information).

See also *pause()*, *sigwait()* and *sigtimedwait()*.

New in version 3.3.

Changed in version 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

`signal.sigtimedwait(`*sigset*, *timeout*`)`
Like *sigwaitinfo()*, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns *None* if a timeout occurs.

Availability: Unix (see the man page *sigtimedwait(2)* for further information).

See also *pause()*, *sigwait()* and *sigwaitinfo()*.

New in version 3.3.

Changed in version 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

19.8.3 Example

Here is a minimal example program. It uses the *alarm()* function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the *os.open()* to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

19.8.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a `SIGPIPE` signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set `SIGPIPE`'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.

19.9 mmap — Memory-mapped file support

Memory-mapped file objects behave like both *bytearray* and like *file objects*. You can use mmap objects in most places where *bytearray* are expected; for example, you can use the *re* module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the *mmap* constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the *os.open()* function, which returns a file descriptor directly (the file still needs to be closed when done).

Note: If you want to create a memory-mapping for a writable, buffered file, you should *flush()* the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of four values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively, or `ACCESS_DEFAULT` to defer to *prot*. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a *TypeError* exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

Changed in version 3.7: Added `ACCESS_DEFAULT` constant.

To map anonymous memory, -1 should be passed as the fileno along with the length.

class mmap.mmap(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is `0`, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the `ALLOCATIONGRANULARITY`.

class mmap.mmap(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when *mmap* is called.

flags specifies the nature of the mapping. `MAP_PRIVATE` creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and `MAP_SHARED` creates a

mapping that's shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. *prot* defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of ALLOCATIONGRANULARITY which is equal to PAGESIZE on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap can also be used as a context manager in a with statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

New in version 3.2: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

```

(continues on next page)

(continued from previous page)

```
mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

Memory-mapped file objects support the following methods:

close()
Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed
`True` if the file is closed.

New in version 3.2.

find(*sub*[, *start*[, *end*]])
Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

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

flush([*offset*[, *size*]])
Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

move(*dest*, *src*, *count*)
Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a *TypeError* exception.

read([*n*])
Return a *bytes* containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Changed in version 3.3: Argument can be omitted or `None`.

read_byte()
Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(*newsize*)
Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or

ACCESS_COPY, resizing the map will raise a *TypeError* exception.

rfind(*sub*[, *start*[, *end*]])
Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

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

seek(*pos*[, *whence*])
Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(*bytes*)
Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a *ValueError* will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will raise a *TypeError* exception.

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

Changed in version 3.6: The number of bytes written is now returned.

write_byte(*byte*)
Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with ACCESS_READ, then writing to it will raise a *TypeError* exception.