18.5.5. Streams (coroutine based API)

Source code: Lib/asyncio/streams.py

18.5.5.1. Stream functions

Note: The top-level functions in this module are meant as convenience wrappers only; there's really nothing special there, and if they don't do exactly what you want, feel free to copy their code.

coroutine `asyncio.`**`open_connection`**(*host=None*, *port=None*, *, *loop=None*, *limit=None*, ***kwds*)

A wrapper for `create_connection()` returning a (reader, writer) pair.

The reader returned is a `StreamReader` instance; the writer is a `StreamWriter` instance.

The arguments are all the usual arguments to `AbstractEventLoop.create_connection()` except *protocol_factory*; most common are positional host and port, with various optional keyword arguments following.

Additional optional keyword arguments are *loop* (to set the event loop instance to use) and *limit* (to set the buffer limit passed to the `StreamReader`).

This function is a coroutine.

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

Start a socket server, with a callback for each client connected. The return value is the same as `create_server()`.

The *client_connected_cb* parameter is called with two parameters: *client_reader*, *client_writer*. *client_reader* is a `StreamReader` object, while *client_writer* is a `StreamWriter` object. The *client_connected_cb* parameter can either be a plain callback function or a coroutine function; if it is a coroutine function, it will be automatically converted into a `Task`.

The rest of the arguments are all the usual arguments to `create_server()` except *protocol_factory*; most common are positional *host* and *port*, with various optional keyword arguments following.

Additional optional keyword arguments are *loop* (to set the event loop instance to use) and *limit* (to set the buffer limit passed to the `StreamReader`).

This function is a coroutine.

coroutine `asyncio.`**`open_unix_connection`**(*path=None*, *, *loop=None*, *limit=None*, ***kwds*)

A wrapper for `create_unix_connection()` returning a (reader, writer) pair.

See `open_connection()` for information about return value and other details.

This function is a coroutine.

Availability: UNIX.

coroutine `asyncio.`**`start_unix_server`**(*client_connected_cb*, *path=None*, *, *loop=None*, *limit=None*, ***kwds*)

Start a UNIX Domain Socket server, with a callback for each client connected.

See `start_server()` for information about return value and other details.

This function is a coroutine.

Availability: UNIX.

18.5.5.2. StreamReader

class `asyncio.`**`StreamReader`**(*limit=None*, *loop=None*)

This class is not thread safe.

`exception`()

Get the exception.

`feed_eof`()

Acknowledge the EOF.

`feed_data`(*data*)

Feed *data* bytes in the internal buffer. Any operations waiting for the data will be resumed.

`set_exception`(*exc*)

Set the exception.

`set_transport`(*transport*)

Set the transport.

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

This method is a coroutine.

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 will return the partial read bytes.

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

This method is a coroutine.

coroutine **`readexactly`**(*n*)

Read exactly *n* bytes. Raise an `IncompleteReadError` if the end of the stream is reached before *n* can be read, the `IncompleteReadError.partial` attribute of the exception contains the partial read bytes.

This method is a coroutine.

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.

Configured stream limit is used to check result. Limit sets the maximal length of data that can be returned, not counting the separator.

If an EOF occurs and the complete separator is still not found, an `IncompleteReadError` exception will be raised, and the internal buffer will be reset. The `IncompleteReadError.partial` attribute may contain the separator partially.

If the data cannot be read because of over limit, a `LimitOverrunError` exception will be raised, and the data will be left in the internal buffer, so it can be read again.

New in version 3.5.2.

`at_eof`()

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

18.5.5.3. StreamWriter

class `asyncio.`**`StreamWriter`**(*transport*, *protocol*, *reader*, *loop*)

Wraps a Transport.

This exposes `write()`, `writelines()`, `can_write_eof()`, `write_eof()`, `get_extra_info()` and `close()`. It adds `drain()` which returns an optional `Future` on which you can wait for flow control. It also adds a transport attribute which references the `Transport` directly.

This class is not thread safe.

`transport`

Transport.

`can_write_eof`()

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

`close`()

Close the transport: see `BaseTransport.close()`.

coroutine **`drain`**()

Let the write buffer of the underlying transport a chance to be flushed.

The intended use is to write:

```
w.write(data)
yield from w.drain()
```

When the size of the transport buffer reaches the high-water limit (the protocol is paused), block until the size of the buffer is drained down to the low-water limit and the protocol is resumed. When there is nothing to wait for, the yield-from continues immediately.

Yielding from `drain()` gives the opportunity for the loop to schedule the write operation and flush the buffer. It should especially be used when a possibly large amount of data is written to the transport, and the coroutine does not yield-from between calls to `write()`.

This method is a coroutine.

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

Return optional transport information: see `BaseTransport.get_extra_info()`.

`write`(*data*)

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

`writelines`(*data*)

Write a list (or any iterable) of data bytes to the transport: see `WriteTransport.writelines()`.

`write_eof`()

Close the write end of the transport after flushing buffered data: see `WriteTransport.write_eof()`.

18.5.5.4. StreamReaderProtocol

class `asyncio.`**`StreamReaderProtocol`**(*stream_reader*, *client_connected_cb=None*, *loop=None*)

Trivial helper class to adapt between `Protocol` and `StreamReader`. Subclass of `Protocol`.

stream_reader is a `StreamReader` instance, *client_connected_cb* is an optional function called with (stream_reader, stream_writer) when a connection is made, *loop* is the event loop instance to use.

(This is a helper class instead of making `StreamReader` itself a `Protocol` subclass, because the `StreamReader` has other potential uses, and to prevent the user of the `StreamReader` from accidentally calling inappropriate methods of the protocol.)

18.5.5.5. IncompleteReadError

exception `asyncio.`**`IncompleteReadError`**

Incomplete read error, subclass of `EOFError`.

`expected`

Total number of expected bytes (`int`).

`partial`

Read bytes string before the end of stream was reached (`bytes`).

18.5.5.6. LimitOverrunError

exception `asyncio.`**`LimitOverrunError`**

Reached the buffer limit while looking for a separator.

`consumed`

Total number of to be consumed bytes.

18.5.5.7. Stream examples

18.5.5.7.1. TCP echo client using streams

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

```python
import asyncio

@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1', 8
                                                        loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = yield from reader.read(100)
    print('Received: %r' % data.decode())

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

message = 'Hello World!'
loop = asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

See also: The TCP echo client protocol example uses the `AbstractEventLoop.create_connection()` method.

18.5.5.7.2. TCP echo server using streams

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

```
import asyncio

@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    writer.write(data)
    yield from writer.drain()

    print("Close the client socket")
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

See also: The TCP echo server protocol example uses the `AbstractEventLoop.create_server()` method.

18.5.5.7.3. Get HTTP headers

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

```
import asyncio
import urllib.parse
import sys

@asyncio.coroutine
def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        connect = asyncio.open_connection(url.hostname, 443, ssl=True)
    else:
        connect = asyncio.open_connection(url.hostname, 80)
    reader, writer = yield from connect
    query = ('HEAD {path} HTTP/1.0\r\n'
             'Host: {hostname}\r\n'
             '\r\n').format(path=url.path or '/', hostname=url.hostnam
    writer.write(query.encode('latin-1'))
    while True:
        line = yield from reader.readline()
        if not line:
            break
        line = line.decode('latin1').rstrip()
        if line:
            print('HTTP header> %s' % line)

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

url = sys.argv[1]
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(print_http_headers(url))
loop.run_until_complete(task)
loop.close()
```

Usage:

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

or with HTTPS:

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

18.5.5.7.4. Register an open socket to wait for data using streams

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

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

@asyncio.coroutine
def wait_for_data(loop):
    # Create a pair of connected sockets
    rsock, wsock = socketpair()
    # Register the open socket to wait for data
    reader, writer = yield from asyncio.open_connection(sock=rsock, lo

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

    # Wait for data
    data = yield from reader.read(100)

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

    # Close the second socket
    wsock.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(wait_for_data(loop))
loop.close()
```

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

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