18. Asynchronous Alternatives - Python in a Nutshell, 3rd Edition

asyncio's Event Loop

`asyncio` supplies an explicit *event loop* interface, and some basic implementations of that interface, as by far the largest component of the framework.

The event loop interface is a very broad and rich interface supplying several categories of methods; the interface is embodied in the `asyncio.BaseEventLoop` class. `asyncio` offers a few alternative implementations of event loops, depending on your platform, and third-party add-on packages let you add still more, for example, to integrate with specific user interface frameworks such as `Qt`. The core idea is that all event loop implementations implement the same interface.

In theory, your application can have multiple event loops and manage them through multiple, explicit *event loop policies*; in practice, such complexity is rarely needed, and you can get away with a single event loop, normally in your main thread, managed by a single, implicit global policy. The main exception to this state of things occurs when you want to run `asyncio` code on multiple threads of your program; in that case, while you may still get away with a single policy, you need multiple event loops—a separate one per each thread calling event-loop methods other than `call_soon_threadsafe`, covered in Table 18-1. Each thread can only call methods on a loop instantiated and running in that thread.

In the rest of this chapter, we assume (except when we explicitly state otherwise) that you're using a single event loop, usually the default one obtained by calling `loop = asyncio.get_event_loop()` near the start of your code. Sometimes, you force a specific implementation, by first instantiating `loop` explicitly as your specific platform or third-party framework may require, then immediately calling `asyncio.set_event_loop(loop)`; ignoring platform-specific or framework-specific semantic peculiarities, which we don't cover in this book, the material in the rest of this chapter applies equally well in this second, somewhat-rarer use case.

The following sections cover nine categories of methods supplied by an `asyncio.BaseEventLoop` instance `loop`, and a few closely related functions and classes supplied by `asyncio`.

Loop state and debugging mode

`loop` can be in one of three states: *stopped* (that's the state `loop` is when just created: nothing yet runs on `loop`), *running* (all functionality runs on `loop`), or *closed* (`loop` is irreversibly terminated, and cannot be started again). Independently, `loop` can be in *debug* mode (checking sanity of operations, and giving ample information to help you develop code) or not (faster and quieter operation; that's the normal mode to use "in production," as opposed to development, debugging, and testing), as covered in "asyncio developing and debugging". Regarding state and mode, `loop` supplies the following methods:

close	`close()`	Sets `loop` to closed state; loses pending callbacks, flushes queues, asks `loop`'s executor to shut down. You can call `loop.close()` only when `loop.is_running()` is `False`. After `loop.close()`, call no further methods on `loop`, except `loop.is_closed()` or `loop.is_closing()` (both return `True` in this case) and `loop.close()` (which does nothing in this case).
get_debug	`get_debug()`	Returns `True` when `loop` is in debug mode; otherwise, returns `False`. The initial value is `True` when the environment variable `PYTHONASYNCIODEBUG` is a nonempty string; otherwise, `False`.
is_closed	`is_closed()`	Returns `True` when `loop` is closed; otherwise, returns `False`.
is_closing	`is_closing()`	Returns `True` when `loop` is closing or already closed; otherwise, returns `False`.
is_running	`is_running()`	Returns `True` when `loop` is running; otherwise, returns `False`.
run_forever	`run_forever()`	Runs `loop` until `loop.stop()` is called. Returns, eventually, after a call to `stop` has placed `loop` in stopped mode.
run_until_complete	`run_until_complete(`*future*`)`	Runs until `future` (an instance of `asyncio.Future`, covered in "Futures") completes; if `future` is a coroutine object or other awaitable, it's wrapped with `asyncio.ensure_future`, covered in "Tasks". When `future` is done, `run_until_complete` returns `future`'s result or raises its exception.
set_debug	`set_debug(`*debug*`)`	Sets `loop`'s debug mode to `bool(`*debug*`)`.
stop	`stop()`	If called when `loop` is stopped, `stop` polls the I/O selector once, with a `timeout` of zero; runs all callbacks scheduled (previously, or in response to I/O events that occurred in that one-off poll of the I/O selector); then exits and leaves `loop` in the stopped state. If called when `loop` is running (e.g., in `run_forever`), `stop` runs the currently scheduled callbacks, then exits and leaves `loop` in the stopped state (in this case, callbacks newly scheduled by other callbacks don't execute; they remain queued, to execute when `run_forever` is called again). After this, the `run_forever` that had put `loop` in running mode returns to its caller.

asyncio developing and debugging

When you develop code using `asyncio`, sanity checking and logging help a lot.

Besides calling `loop.set_debug(True)` (or setting the environment variable `PYTHONASYNCIODEBUG` to a nonempty string), set logging to `DEBUG` level: for example, call `logging.basicConfig(level=logging.DEBUG)` at startup.

In debug mode, you get useful `ResourceWarning` warnings when transports and event loops are not explicitly closed (frequently a symptom of a bug in your code): enable warnings, as covered in "The warnings Module"—for example, use the command-line option `-Wdefault` (with no space between `-W` and `default`) when you start Python, as mentioned in Table 2-1.

For more tips and advice on developing and debugging with `asyncio`, see the appropriate section in the online docs.

Calls, time, and sleep

A key functionality of the `asyncio` event loop is to schedule calls to functions (see Table 18-1)—either "as soon as convenient" or with specified delays. For the latter purpose, *loop* maintains its own internal clock (in seconds and fractions), not necessarily coincident with the system clock covered in Chapter 12.

Use functools.partial when you need to pass named arguments in calls

loop's methods to schedule calls don't directly support passing named arguments. If you do need to pass named arguments, wrap the function to be called in `functools.partial`, covered in Table 7-4. This is the best way to achieve this goal (superior to alternatives such as using `lambda` or closures), because debuggers (including `asyncio`'s debug mode) introspect such wrapped functions to supply more and clearer information than they could with alternative approaches.

Table 18-1.

call_at	`call_at(when, callback, *args)` Schedules `callback(*args)` to be called when `loop.time()` equals `when` (or, as soon as feasible after that, should something else be running at that precise time). Returns an instance of `asyncio.Handle`, on which you can call the method `cancel()` to cancel the callback (innocuous if the call has already occurred).
call_later	`call_later(delay, callback, *args)` Schedules `callback(*args)` to be called `delay` seconds from now (`delay` can include a fractional part) (or, as soon as feasible after that, should something else be running at that precise time). Returns an instance of `asyncio.Handle`, on which you can call method `cancel()` to cancel the callback (innocuous if the call has already occurred).
call_soon	`call_soon(callback, *args)` Schedules `callback(*args)` to be called as soon as possible (in first-in, first-out order with regard to other scheduled callbacks). Returns an instance of `asyncio.Handle`, on which you can call method `cancel()` to cancel the callback (innocuous if the call has already occurred).

call_soon_threadsafe	`call_soon_threadsafe(`*`callback`*`, *`*`args`*`)` Like `call_soon`, but safe to call from a thread different from the one `loop` is running in (`loop` should normally run in the main thread, to allow signal handling and access from other processes).
time	`time()` Returns a `float` that is `loop`'s current internal time.

Besides *`loop`*'s own methods, the current event loop's internal time is also used by one module-level function (i.e., a function supplied directly by the module `asyncio`):

sleep	`coroutine sleep(`*`delay`*`, `*`result`*`=None)` A coroutine function to build and return a coroutine object that completes after `delay` seconds and returns `result` (`delay` can include a fractional part).

Connections and server

`loop` can have open communication channels of several kinds: connections that reach out to other systems' listening sockets (stream or datagram or—on Unix—unix sockets), ones that listen for incoming connections (grouped in instances of the `Server` class), and ones built on pipes to/from subprocesses. Here are the methods `loop` supplies to create the various kinds of connections:

create_connection

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

`protocol_factory`, the one argument that's always required, is a callable taking no arguments and returning a protocol instance. `create_connection` is a coroutine function: its resulting coroutine object, as `loop` runs it, creates the connection if required, wraps it into a transport instance, creates a protocol instance, ties the protocol instance to the transport instance with the protocol's `connection_made` method, and finally—once all is done—returns a pair `(transport, protocol)` as covered in "Transports and protocols".

When you have an already-connected stream socket that you just want to wrap into a transport and protocol, pass it as the named argument `sock`, and avoid passing any of `host`, `port`, `family`, `proto`, `flags`, `local_addr`; otherwise, `create_connection` creates and connects a new stream socket for you (family `AF_INET` or `AF_INET6` depending on `host`, or `family` if explicitly specified; type `SOCK_STREAM`), and you need to pass some or all of those arguments to specify the details of the socket to connect.

Optional arguments must be passed as named arguments, if at all.

`ssl`, when true, requests an SSL/TLS transport (in particular, when it's an instance of `ssl.SSLContext`, as covered in "SSLContext", that instance is used to create the transport); in this case, you may optionally pass `server_hostname` as the hostname to match the server's certificate against (or an empty string, disabling hostname matching, but that's a serious security risk). When you specify `host`, you may omit `server_hostname`; in this case, `host` is the hostname that must match the certificate.

All other optional named-only arguments can be passed only if `sock` is not passed. `family`, `proto`, and `flags` are passed through to `getaddrinfo` to resolve the `host`; `local_addr` is a `(local_host, local_port)` pair passed through to `getaddrinfo` to resolve the local address to which to bind the socket being created.

create_datagram_endpoint

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

Much like `create_connection`, except that the connection is a datagram rather than stream one (i.e., socket type is `SOCK_DGRAM`). `remote_addr` can be optionally be passed as a `(remote_host, remote_port)` pair. `reuse_address`, when true (the default), means to reuse a local socket without waiting for its natural timeout to expire. `reuse_port`, when true, on some Unix-like systems, allows multiple datagram connections to be bound to the same port. `allow_broadcast`, when true, allows this connection to send datagrams to the broadcast address.

create_server	`coroutine 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)` Like `create_connection`, except that it creates or wraps a *listening* socket and returns an instance of the class `asyncio.Server` (with the attribute `sockets`, the list of `socket` objects the server is listening to; the method `close()` to close all sockets asynchronously; and the coroutine method `wait_closed()` to wait until all sockets are closed). `host` can be a string, or a sequence of strings in order to bind multiple hosts; when `None`, it binds all interfaces. `backlog` is the maximum number of queued connections (passed on to the underlying socket's `listen` method). `reuse_address`, when true, means to reuse a local socket without waiting for its natural timeout to expire. `reuse_port`, when true, on some Unix-like systems, allows multiple listening connections to be bound to the same port.
create_unix_connection	`coroutine create_unix_connection(protocol_factory, path, *, ssl=None, sock=None, server_hostname=None)` Like `create_connection`, except that the connection is on a Unix socket (socket family `AF_UNIX`, socket type `SOCK_STREAM`) on the given `path`. Unix sockets allow very fast, secure communication, but only between processes on a single Unix-like computer.
create_unix_server	`coroutine create_unix_server(protocol_factory, path=None, *, sock=None, backlog=100, ssl=None)` Same as `create_server`, but for Unix-socket connections on the given `path`.

Besides sockets, event loops can connect subprocess pipes, using these methods:

connect_read_pipe	`coroutine connect_read_pipe(protocol_factory, pipe)` Returns a `(transport, protocol)` pair wrapping read-mode, nonblocking file-like object `pipe`.
connect_write_pipe	`coroutine connect_write_pipe(protocol_factory, pipe)` Returns a `(transport, protocol)` pair wrapping write-mode, nonblocking file-like object `pipe`.

Tasks

An `asyncio` task (`asyncio.Task`, covered in "Tasks", is a subclass of `asyncio.Future`, covered in "Futures") wraps a coroutine object and orchestrates its execution. `loop` offers a method to create a task:

create_task	`create_task(coro)` Creates and returns a `Future` (usually, a `Task`) wrapping the coroutine object `coro`.

You can also customize the factory `loop` uses to create tasks, but that is rarely needed except to write custom implementations of event loops, so we don't cover it.

Another roughly equivalent way to create a task is to call `asyncio.ensure_future` with a single argument that is a coroutine object or other awaitable; the function in this case creates and returns a `Task` instance wrapping the coroutine object. (If you call `asyncio.ensure_future` with an argument that's a `Future` instance, it returns the argument unchanged.)

Create tasks with loop.create_task

We recommend using more explicit and readable `loop.create_task` instead of the roughly equivalent `asyncio.ensure_future`.

Watching file descriptors

You can choose to use `loop` at a somewhat-low abstraction level, watching for file descriptors to become ready for reading or writing, and calling callback functions when they do. (On Windows, with the default `SelectorEventLoop` implementation of `loop`, you can use these methods only on file descriptors representing sockets; with the alternative `ProactorEventLoop` implementation that you can choose to explicitly instantiate, you cannot use these methods at all.) `loop` supplies the following methods related to watching file descriptors:

add_reader	`add_reader(fd, callback, *args)` When `fd` becomes available for reading, call `callback(*args)`.
add_writer	`add_writer(fd, callback, *args)` When `fd` becomes available for writing, call `callback(*args)`.
remove_reader	`remove_reader(fd)` Stop watching for `fd` to become available for reading.
remove_writer	`remove_writer(fd)` Stop watching for `fd` to become available for writing.

socket operations; hostnames

Also at a low level of abstraction, *`loop`* supplies four coroutine-function methods corresponding to methods on socket objects covered in Chapter 17:

sock_accept	`coroutine sock_accept(sock)` `sock` must be nonblocking, bound to an address, and listening for connections. `sock_accept` returns a coroutine object that, when done, returns a pair `(conn, address)`, where `conn` is a new socket object to send and receive data on the connection and `address` is the address bound to the socket of the counterpart.

sock_connect	`coroutine sock_connect(sock, address)`	`sock` must be nonblocking and not already connected. `address` must be the result of a `getaddrinfo` call, so that `sock_connect` doesn't have to use DNS itself (for example, for network sockets, `address` must include an IP address, not a hostname). `sock_connect` returns a coroutine object that, when done, returns `None` and leaves `sock` appropriately connected as requested.
sock_recv	`coroutine sock_recv(sock, nbytes)`	`sock` must be nonblocking and connected. `nbytes` is an `int`, the maximum number of bytes to receive. `sock_recv` returns a coroutine object that, when done, returns a `bytes` object with the data received on the socket (or raises an exception if a network error occurs).
sock_sendall	`coroutine sock_sendall(sock, data)`	`sock` must be nonblocking and connected. `sock_sendall` returns a coroutine object that, when done, returns `None`, having sent all the bytes in `data` on the socket. (In case of a network error, an exception is raised; there is no way to determine how many bytes, if any, were sent to the counterpart before the network error occurred.)

It's often necessary to perform DNS lookups. `loop` supplies two coroutine-function methods that work like the same-name functions covered in Table 17-1, but in an async, nonblocking way:

getaddrinfo	`coroutine getaddrinfo(host, port, *, family=0, type=0, proto=0, flags=0)`	Returns a coroutine object that, when done, returns a five-items tuple `(family, type, proto, canonname, sockaddr)`, like `socket.getaddrinfo`.
getnameinfo	`coroutine getnameinfo(sockaddr, flags=0)`	Returns a coroutine object that, when done, returns a pair `(host, port)`, like `socket.getnameinfo`.

Unix signals

On Unix-like platforms, `loop` (when run on the main thread) supplies two methods to add and remove handlers for *signals* (a Unix-specific, limited form of inter-process communication, well covered on Wikipedia) the process may receive:

add_signal_handler	`add_signal_handler(signum, callback, *args)`	Sets the handler for signal number `signum` to call `callback(*args)`.
remove_signal_handler	`remove_signal_handler(signum)`	Removes the handler for signal number `signum`, if any. Returns `True` when it removes a handler, and `False` when there was no handler to remove (in either case, `signum` now has no handler).

Executor

`loop` can arrange for a function to run in an *executor*, a pool of threads or processes as covered in "The concurrent.futures Module": that's useful when you must do some blocking I/O, or CPU-intensive operations (in the latter case, use as executor an instance of `concurrent.futures.ProcessPoolExecutor`). The two relevant methods are:

run_in_executor	`run_in_executor(`*`executor, func,`*` coroutine *`*`args`*`)` Returns a coroutine object that runs `func(*args)` in `executor` and, when done, returns `func`'s result. `executor` must be an `Executor` instance, or `None`, to use `loop`'s current default executor.
set_default_executor	`set_default_executor(`*`executor`*`)` Sets `loop`'s executor to `executor`, which must be an `Executor` instance, or `None`, to use the default (thread pool) executor.

Error handling

You can customize exception handling in the event loop; `loop` supplies four methods for this purpose:

call_exception_handler	`call_exception_handler(`*`context`*`)` Call `loop`'s current exception handler.
default_exception_handler	`default_exception_handler(`*`context`*`)` The exception handler supplied by `loop`'s class; it's called when an exception occurs and no handler is set, and may be called by a handler to defer a case to the default behavior.
get_exception_handler	`get_exception_handler()` Gets and returns `loop`'s current exception handler, a callable accepting two arguments, `loop` and `context`.
set_exception_handler	`set_exception_handler(`*`handler`*`)` Sets `loop`'s current exception handler to `handler`, a callable accepting two arguments, `loop` and `context`.

`context` is a `dict` with the following contents (more keys may be added in future releases). All keys, except `message`, are optional; use `context.get(key)` to avoid a `KeyError` on accessing some key in the context:

exception

`Exception` instance

future

`asyncio.Future` instance

handle

`asyncio.Handle` instance

message

`str` instance, the error message

protocol

`asyncio.Protocol` instance

socket

`socket.socket` instance

transport

`asyncio.Transport` instance

The following sections cover three more concepts you need in order to use `asyncio`, and functionality supplied by `asyncio` for each of these concepts.

Futures

The `asyncio.Future` class is almost compatible with the `Future` class supplied by the module `concurrent.futures` and covered in Table 14-1 (in some future version of Python, the intention is to fully unify the two `Future` interfaces, but this goal cannot be guaranteed). The main differences between an instance `af` of `asyncio.Future` and an instance `cf` of `concurrent.futures.Future` are:

- `af` is not thread-safe
- `af` can't be passed to functions `wait` and `as_completed` of module `concurrent.futures`
- Methods *`af`*`.result` and *`af`*`.exception` don't take a `timeout` argument, and can only be called when *`af`*`.done()` is `True`
- There is no method *`af`*`.running()`

For thread-safety, callbacks added with *`af`*`.add_done_callback` get scheduled, once `af` is done, via *`loop`*`.call_soon_threadsafe`.

`af` also supplies three extra methods over and above those of `cf`:

remove_done_callback	`remove_done_callback(`*`func`*`)` Removes all instances of `func` from the list of `af`'s callbacks; returns the number of instances it removed.
set_exception	`set_exception(`*`exception`*`)` Marks `af` done, and set its exception to `exception`. If `af` is already done, `set_exception` raises an exception.
set_result	`set_result(`*`value`*`)` Marks `af` done, and sets its result to *`value`*. If `af` is already done, `set_result` raises an exception.

(In fact, `cf` has methods `set_exception` and `set_result`, too, but in `cf`'s case they're meant to be called strictly and exclusively by unit tests and `Executor` implementations; `af`'s identical methods do not have such constraints.)

The best way to create a `Future` in `asyncio` is with `loop`'s `create_future` method, which takes no arguments; at worst, *`loop`*`.create_future()` just performs exactly the same as `return futures.Future(`*`loop`*`)`, but, this way, alternative loop implementations get a chance to override the method and provide a better implementation of futures.

Tasks

`asyncio.Task` is a subclass of `asyncio.Future`: an instance `at` of `asyncio.Task` wraps a coroutine object and schedules its execution in `loop`.

The class defines two class methods: `all_tasks()`, which returns the `set` of all tasks defined on `loop`; and `current_task()`, which returns the task currently executing (`None` if no task is executing).

`at``.cancel()` has slightly different semantics from the `cancel` method of other futures: it does not guarantee the cancellation of the task, but rather raises a `CancelledError` inside the wrapped coroutine—the latter may intercept the exception (intended to enable clean-up work, but also makes it possible for the coroutine to refuse cancellation). *`at`*`.cancelled()` returns `True` only when the wrapped coroutine has propagated (or spontaneously raised) `CancelledError`.

`asyncio` supplies several functions to ease working with tasks and other futures. All accept an optional (named-only) argument `loop=None` to use an event loop different from the current default one (`None` means to use the current default event loop); for all, arguments that are futures can also be coroutine objects (which get automatically wrapped into instances of `asyncio.Task`). The functions are:

as_completed	`as_completed(`*`futures`*`, *, loop=None, timeout=None)` Returns an iterator whose values are `Future` instances (yielded approximately in order of completion). When `timeout` is not `None`, it's a value in seconds (which may have a fractional part), and in that case `as_completed` raises `asyncio.TimeOuterror` after `timeout` seconds unless all futures have completed by then.
gather	`gather(*`*`futures`*`, *, loop>=None, return_exceptions=False)` Returns a single future `f` whose result is a list of the results of the `futures` arguments, in the same order, when all have completed (all must be futures in the same event loop). When `return_exceptions` is false, any exception raised in a contained future immediately propagates through `f`; when `return_exceptions` is true, contained-future exceptions are put in `f`'s result list, just like contained-future results. *`f`*`.cancel()` cancels any contained future that's not yet done. If any contained future is separately cancelled, that's just as if it had raised a `CancelledError` (therefore, this does not cancel `f`, as long as `return_exceptions` is true).

run_coroutine_threadsafe

`run_coroutine_threadsafe(coro, loop)`

Submits coroutine object `coro` to event loop `loop`, returns a `concurrent.futures.Future` instance to access the result. This function is meant to allow other threads to safely submit coroutine objects to `loop`.

shield

`shield(f, *, loop=None)`

Waits for `Future` instance *f*, shielding `f` against cancellation if the coroutine doing `await asyncio.shield(f)` (or `yield from asyncio.shield(f)`) is cancelled.

timeout

`timeout(timeout, *, loop=None)`

Returns a context manager that raises an `asyncio.TimeoutError` if a block has not completed after `timeout` seconds (`timeout` may have a fractional part). For example:

```
try:
    with asyncio.timeout(0.5):
        await first()
        await second()
except asyncio.TimeoutError:
    print('Alas, too slow!')
else:
    print('Made it!')
```

This snippet prints `Made it!` when the two awaitables `first()` and `second()`, in sequence, both complete within half a second; otherwise, it prints `Alas, too slow!`

wait

coroutine `wait(futures, *, loop=None, timeout=None, return_when=ALL_COMPLETED)`

This coroutine function returns a coroutine object that waits for the futures in nonempty iterable `futures`, and returns a tuple of two `set`s of futures, `(done, still_pending)`. `return_when` must be one of three constants defined in module `concurrent.futures`: `ALL_COMPLETED`, the default, returns when all futures are done (so the returned `still_pending set` is empty); `FIRST_COMPLETED` returns as soon as any of the futures is done; `FIRST_EXCEPTION` is like `ALL_COMPLETED` but also returns if and when any future raises an exception (in which case `still_pending` may be nonempty).

wait_for	`wait_for(f, timeout, *,` coroutine `loop=None)`	This coroutine function returns a coroutine object that waits for future `f` for up to `timeout` seconds (`timeout` may have a fractional part, or may be `None`, meaning to wait indefinitely) and returns `f`'s result (or raises `f`'s exception, or `asyncio.TimeoutError` if a timeout occurs).

Transports and protocols

For details about transports and protocols, see the section about them in the online docs. In this section, we're offering just the conceptual basis, some core details about working with them, and two examples. The core idea is that a *transport* does all that's needed to ensure that a stream (or datagram) of "raw," uninterpreted bytes is pushed to an external system, or pulled from an external system; a *protocol* translates those bytes to and from semantically meaningful messages.

A *transport* class is a class supplied by `asyncio` to abstract any one of various kinds of communication channels (TCP, UDP, SSL, pipes, etc.). You don't directly instantiate a transport class: rather, you call `loop` methods that create the transport instance and the underlying channel, and provide the transport instance when done.

A *protocol* class is one supplied by `asyncio` to abstract various kinds of protocols (streaming, datagram-based, subprocess-pipes). Extend the appropriate one of those base classes, overriding the callback methods in which you want to perform some action (the base classes supply empty default implementations for such methods, so just don't override methods for events you don't care about). Then, pass your class as the `protocol_factory` argument to `loop` methods.

A protocol instance `p` always has an associated transport instance `t`, in 1-to-1 correspondence. As soon as the connection is established, `loop` calls `p.connection_made(t)`: `p` must save `t` as an attribute of `self`, and may perform some initialization-setting method calls on `t`.

When the connection is lost or closed, `loop` calls `p.connection_lost(exc)`, where `exc` is `None` to indicate a regular closing (typically via end-of-file, EOF), or else an `Exception` instance recording what error caused the connection to be lost.

Each of `connection_made` and `connection_lost` gets called exactly once on each protocol instance `p`. All other callbacks to `p`'s methods happen between those two calls; during such other callbacks, `p` gets informed by `t` about data or EOF being received, and/or asks `t` to send data out. All interactions between `p` and `t` occur via callbacks by each other on the other one's methods.

Protocol-based examples: echo client and server

Here is a protocol-based implementation of a client for the same simple echo protocol shown in "A Connection-Oriented Socket Client". (Since `asyncio` exists only in v3, we have not bothered maintaining any compatibility with v2 in this example's code.)

```
import asyncio

       """A few lines of
data = text
including non-ASCII characters:
€£
to test the
operation
of both
server
and
client."""

class EchoClient(asyncio.Protocol):

  def __init__(self):
      self.data_iter = iter(data.splitlines())

  def write_one(self):
      chunk = next(self.data_iter, None)
      if chunk is None:
          self.transport.write_eof()
      else:
          line = chunk.encode()
          self.transport.write(line)
          print('Sent:', chunk)

  def connection_made(self, transport):
      self.transport = transport
            'Connected to
      print(server'              )
      self.write_one()

  def connection_lost(self, exc):
      loop.stop()
      print('Disconnected from server')

  def data_received(self, data):
      print('Recv:', data.decode())
      self.write_one()

loop = asyncio.get_event_loop()
echo = loop.create_connection(EchoClient, 'localhost', 8881)
transport, protocol = loop.run_until_complete(echo)
loop.run_forever()
loop.close()
```

You wouldn't normally bother using `asyncio` for such a simplistic client, one that is doing nothing beyond sending data to the server, receiving replies, and using `print` to show what's happening. However, the purpose of the example is to show how to use `asyncio` (and, specifically, `asyncio`'s protocols) in a client (which would be handy if a client had to communicate with multiple servers and/or perform other nonblocking I/O operations simultaneously).

Nevertheless, this example, for conciseness, takes shortcuts (such as calling `loop.stop` when connection is lost) that would not be acceptable in high-quality production code. For a critique of simplistic echo examples, and a thoroughly productionized counterexample, see Łukasz Langa's aioecho.

Similarly, here is a v3-only protocol-based server for the same (deliberately simplistic) echo functionality:

```
import asyncio

class EchoServer(asyncio.Protocol):

  def connection_made(self, transport):
      self.transport = transport
      self.peer = transport.get_extra_info('peername')
      print('Connected from', self.peer)

  def connection_lost(self, exc):
      print('Disconnected from', self.peer)

  def data_received(self, data):
      print('Recv:', data.decode())
      self.transport.write(data)
      print('Echo:', data.decode())

loop = asyncio.get_event_loop()
echo = loop.create_server(EchoServer, 'localhost', 8881)
server = loop.run_until_complete(echo)
      'Serving
print(at'         , server.sockets[0].getsockname())
loop.run_forever()
```

This server code has no intrinsic limits on how many clients at a time it can be serving, and the transport deals with any network fragmentation.

asyncio streams

Fundamental operations of transports and protocols, as outlined in the previous section, rely on a callback paradigm. As mentioned in "Coroutine-Based Async Architectures" , this may make development harder when you need to fragment what could be a linear stream of code into multiple functions and methods; with `asyncio`, you may partly finesse that by using coroutines, futures, and tasks for implementation—but there is no intrinsic connection between protocol instances and such tools, so you'd essentially be building your own. It's reasonable to wish for a higher level of abstraction, focused directly on coroutines, for the same networking purposes you could use transports, protocols, and their callbacks for.

`asyncio` comes to the rescue by supplying coroutine-based *streams*, as documented online. Four convenience coroutine functions based on streams are directly supplied by `asyncio`: `open_connection`, `open_unix_connection`, `start_server`, and `start_unix_server`. While usable on their own, they're mostly supplied as examples of how to best create streams, and the docs explicitly invite you to copy them into your code and edit them as necessary. You can easily find the source code, for example on GitHub. The same logic applies, and is explicitly documented in source code comments, to the stream classes themselves—`StreamReader` and `StreamWriter`; the classes wrap transports and protocols and supply coroutine methods where appropriate.

Here is a streams-based implementation of a client for the same simple echo protocol shown in "A Connection-Oriented Socket Client", coded using the legacy (pre–Python 3.5) approach to coroutines:

```
import asyncio

       """A few lines of
data = data
including non-ASCII characters:
€£
to test the
operation
of both
server
and
client."""

@asyncio.coroutine
def echo_client(data):
    reader, writer = yield from asyncio.open_connection
(
        'localhost', 8881)
          'Connected to
    print(server'              )
    for line in data:
        writer.write(line.encode())
        print('Sent:', line)
        response = yield from reader.read(1024)
        print('Recv:', response.decode())
    writer.close()
    print('Disconnected from server')

loop = asyncio.get_event_loop()
loop.run_until_complete(echo_client(data.splitlines()))
loop.close()
```

`asyncio.open_connection` eventually (via its coroutine-object immediate result, which `yield from` waits for) returns a pair of streams, `reader` and `writer`, and the rest is easy (with another `yield from` to get the eventual result of `reader.read`). The modern native (Python 3.5) kind of coroutine is no harder:

```
import asyncio

       """A few lines of
data = data
including non-ASCII characters:
€£
to test the
operation
of both
server
and
client."""

async def echo_client(data):
    reader, writer = await asyncio.open_connection(
        'localhost', 8881)
          'Connected to
    print(server'               )
    for line in data:
        writer.write(line.encode())
        print('Sent:', line)
        response = await reader.read(1024)
        print('Recv:', response.decode())
    writer.close()
    print('Disconnected from server')

loop = asyncio.get_event_loop()
loop.run_until_complete(echo_client(data.splitlines()))
loop.close()
```

The transformation needed in this case is purely mechanical: remove the decorator, change `def` to `async def`, and change `yield from` to `await`.

Similarly for streams-based servers—here's one that uses legacy coroutines:

```
import asyncio

@asyncio.coroutine
def handle(reader, writer):
    address = writer.get_extra_info('peername')
    print('Connected from', address)

    while True:
        data = yield from reader.read(1024)
        if not data: break
        s = data.decode()
        print('Recv:', s)
        writer.write(data)
        yield from writer.drain()
        print('Echo:', s)

    writer.close()
    print('Disconnected from', address)

loop = asyncio.get_event_loop()
echo = asyncio.start_server(handle, 'localhost', 8881)
server = loop.run_until_complete(echo)

print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

And here’s the equivalent using modern native coroutines, again with the same mechanical transformation:

```
import asyncio

async def handle(reader, writer):
    address = writer.get_extra_info('peername')
    print('Connected from', address)

    while True:
        data = await reader.read(1024)
        if not data: break
        s = data.decode()
        print('Recv:', s)
        writer.write(data)
        await writer.drain()
        print('Echo:', s)

    writer.close()
    print('Disconnected from', address)

loop = asyncio.get_event_loop()
echo = asyncio.start_server(handle, 'localhost', 8881)
server = loop.run_until_complete(echo)

print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

The same considerations as in the previous section apply to these client and server examples: each client, as it stands, may be considered “overkill” for the very simple task it performs (but is useful to suggest how `asyncio`-based clients for more complex tasks would proceed); each server is intrinsically unbounded in the number of clients it can serve, and immune to any data fragmentation that might have occurred during network transmission.