
AnyIO

Release 2.2.0

Alex Grönholm

Feb 27, 2021

CONTENTS

1	The manual	3
1.1	The basics	3
1.2	Creating and managing tasks	4
1.3	Cancellation and timeouts	5
1.4	Using synchronization primitives	7
1.5	Streams	10
1.6	Using typed attributes	15
1.7	Using sockets and streams	16
1.8	Working with threads	18
1.9	Using subprocesses	21
1.10	Asynchronous file I/O support	22
1.11	Receiving operating system signals	22
1.12	Testing with AnyIO	24
1.13	API reference	26
1.14	Getting help	61
1.15	Reporting bugs	61
1.16	Contributing to AnyIO	61
1.17	Version history	62
	Index	69

AnyIO is a asynchronous compatibility API that allows applications and libraries written against it to run unmodified on [asyncio](#), [curio](#) and [trio](#).

It bridges the following functionality:

- Task groups
- Cancellation
- Threads
- Signal handling
- Asynchronous file I/O
- Subprocesses
- Inter-task synchronization and communication (locks, conditions, events, semaphores, object streams)
- High level networking (TCP, UDP and UNIX sockets)

You can even use it together with native libraries from your selected backend in applications. Doing this in libraries is not advisable however since it limits the usefulness of your library.

AnyIO comes with its own [pytest](#) plugin which also supports asynchronous fixtures. It even works with the popular [Hypothesis](#) library.

1.1 The basics

AnyIO requires Python 3.6.2 or later to run. It is recommended that you set up a [virtualenv](#) when developing or playing around with AnyIO.

1.1.1 Installation

To install AnyIO, run:

```
pip install anyio
```

To install a supported version of [trio](#) or [curio](#), you can use install them as extras like this:

```
pip install anyio[curio]
```

1.1.2 Running async programs

The simplest possible AnyIO program looks like this:

```
from anyio import run

async def main():
    print('Hello, world!')

run(main)
```

This will run the program above on the default backend (asyncio). To run it on another supported backend, say [trio](#), you can use the backend argument, like so:

```
run(main, backend='trio')
```

But AnyIO code is not required to be run via `run()`. You can just as well use the native `run()` function of the backend library:

```
import sniffio
import trio
from anyio import sleep
```

(continues on next page)

(continued from previous page)

```
async def main():
    print('Hello')
    await sleep(1)
    print("I'm running on", sniffio.current_async_library())

trio.run(main)
```

1.1.3 Backend specific options

Asyncio:

- `debug` (bool, default=False): Enables `debug mode` in the event loop
- `use_uvloop` (bool, default=True): Use the faster `uvloop` event loop implementation, if available
- `policy` (`AbstractEventLoopPolicy`, default=None): the event loop policy instance to use for creating a new event loop (overrides `use_uvloop`)

Curio: options covered in the [official documentation](#)

Trio: options covered in the [official documentation](#)

1.1.4 Using native async libraries

AnyIO lets you mix and match code written for AnyIO and code written for the asynchronous framework of your choice. There are a few rules to keep in mind however:

- You can only use “native” libraries for the backend you’re running, so you cannot, for example, use a library written for trio together with a library written for asyncio.
- Tasks spawned by these “native” libraries on backends other than `trio` are not subject to the cancellation rules enforced by AnyIO
- Threads spawned outside of AnyIO cannot use `run_async_from_thread()` to call asynchronous code

1.2 Creating and managing tasks

A *task* is a unit of execution that lets you do many things concurrently that need waiting on. This works so that while you can have any number of tasks, the asynchronous event loop can only run one of them at a time. When the task encounters an `await` statement that requires the task to sleep until something happens, the event loop is then free to work on another task. When the thing the first task was waiting is complete, the event loop will resume the execution of that task on the first opportunity it gets.

Task handling in AnyIO loosely follows the `trio` model. Tasks can be created (*spawned*) using *task groups*. A task group is an asynchronous context manager that makes sure that all its child tasks are finished one way or another after the context block is exited. If a child task, or the code in the enclosed context block raises an exception, all child tasks are cancelled. Otherwise the context manager just waits until all child tasks have exited before proceeding.

Here’s a demonstration:

```
from anyio import sleep, create_task_group, run
```

(continues on next page)

(continued from previous page)

```

async def sometask(num):
    print('Task', num, 'running')
    await sleep(1)
    print('Task', num, 'finished')

async def main():
    async with create_task_group() as tg:
        for num in range(5):
            await tg.spawn(sometask, num)

    print('All tasks finished!')

run(main)

```

1.2.1 Handling multiple errors in a task group

It is possible for more than one task to raise an exception in a task group. This can happen when a task reacts to cancellation by entering either an exception handler block or a `finally:` block and raises an exception there. This raises the question: which exception is propagated from the task group context manager? The answer is “both”. In practice this means that a special exception, `ExceptionGroup` is raised which contains both exception objects. Unfortunately this complicates any code that wishes to catch a specific exception because it could be wrapped in an `ExceptionGroup`.

1.3 Cancellation and timeouts

The ability to cancel tasks is the foremost advantage of the asynchronous programming model. Threads, on the other hand, cannot be forcibly killed and shutting them down will require perfect cooperation from the code running in them.

Cancellation in AnyIO follows the model established by the `trio` framework. This means that cancellation of tasks is done via so called *cancel scopes*. Cancel scopes are used as context managers and can be nested. Cancelling a cancel scope cancels all cancel scopes nested within it. If a task is waiting on something, it is cancelled immediately. If the task is just starting, it will run until it first tries to run an operation requiring waiting, such as `sleep()`.

A task group contains its own cancel scope. The entire task group can be cancelled by cancelling this scope.

1.3.1 Timeouts

Networked operations can often take a long time, and you usually want to set up some kind of a timeout to ensure that your application doesn’t stall forever. There are two principal ways to do this: `move_on_after()` and `fail_after()`. Both are used as asynchronous context managers. The difference between these two is that the former simply exits the context block prematurely on a timeout, while the other raises a `TimeoutError`.

Both methods create a new cancel scope, and you can check the deadline by accessing the `deadline` attribute. Note, however, that an outer cancel scope may have an earlier deadline than your current cancel scope. To check the actual deadline, you can use the `current_effective_deadline()` function.

Here’s how you typically use timeouts:

```

from anyio import create_task_group, move_on_after, sleep, run

```

(continues on next page)

(continued from previous page)

```
async def main():
    async with create_task_group() as tg:
        async with move_on_after(1) as scope:
            print('Starting sleep')
            await sleep(2)
            print('This should never be printed')

        # The cancel_called property will be True if timeout was reached
        print('Exited cancel scope, cancelled =', scope.cancel_called)

run(main)
```

1.3.2 Shielding

There are cases where you want to shield your task from cancellation, at least temporarily. The most important such use case is performing shutdown procedures on asynchronous resources.

To accomplish this, open a new cancel scope with the `shield=True` argument:

```
from anyio import create_task_group, open_cancel_scope, sleep, run

async def external_task():
    print('Started sleeping in the external task')
    await sleep(1)
    print('This line should never be seen')

async def main():
    async with create_task_group() as tg:
        async with open_cancel_scope(shield=True) as scope:
            await tg.spawn(external_task)
            await tg.cancel_scope.cancel()
            print('Started sleeping in the host task')
            await sleep(1)
            print('Finished sleeping in the host task')

run(main)
```

The shielded block will be exempt from cancellation except when the shielded block itself is being cancelled. Shielding a cancel scope is often best combined with `move_on_after()` or `fail_after()`, both of which also accept `shield=True`.

1.3.3 Finalization

Sometimes you may want to perform cleanup operations in response to the failure of the operation:

```
async def do_something():
    try:
        await run_async_stuff()
    except BaseException:
        # (perform cleanup)
        raise
```

In some specific cases, you might only want to catch the cancellation exception. This is tricky because each async framework has its own exception class for that and AnyIO cannot control which exception is raised in the task when it's cancelled. To work around that, AnyIO provides a way to retrieve the exception class specific to the currently running async framework, using `get_cancelled_exc_class()`:

```
from anyio import get_cancelled_exc_class

async def do_something():
    try:
        await run_async_stuff()
    except get_cancelled_exc_class():
        # (perform cleanup)
        raise
```

Warning: Always reraise the cancellation exception if you catch it. Failing to do so may cause undefined behavior in your application.

1.4 Using synchronization primitives

Synchronization primitives are objects that are used by tasks to communicate and coordinate with each other. They are useful for things like distributing workload, notifying other tasks and guarding access to shared resources.

1.4.1 Semaphores

Semaphores are used for limiting access to a shared resource. A semaphore starts with a maximum value, which is decremented each time the semaphore is acquired by a task and incremented when it is released. If the value drops to zero, any attempt to acquire the semaphore will block until another task frees it.

Example:

```
from anyio import create_task_group, create_semaphore, sleep, run

async def use_resource(tasknum, semaphore):
    async with semaphore:
        print('Task number', tasknum, 'is now working with the shared resource')
        await sleep(1)

async def main():
    semaphore = create_semaphore(2)
    async with create_task_group() as tg:
        for num in range(10):
            await tg.spawn(use_resource, num, semaphore)

run(main)
```

1.4.2 Locks

Locks are used to guard shared resources to ensure sole access to a single task at once. They function much like semaphores with a maximum value of 1.

Example:

```
from anyio import create_task_group, create_lock, sleep, run

async def use_resource(tasknum, lock):
    async with lock:
        print('Task number', tasknum, 'is now working with the shared resource')
        await sleep(1)

async def main():
    lock = create_lock()
    async with create_task_group() as tg:
        for num in range(4):
            await tg.spawn(use_resource, num, lock)

run(main)
```

1.4.3 Events

Events are used to notify tasks that something they've been waiting to happen has happened. An event object can have multiple listeners and they are all notified when the event is triggered.

Example:

```
from anyio import create_task_group, create_event, run

async def notify(event):
    await event.set()

async def main():
    event = create_event()
    async with create_task_group() as tg:
        await tg.spawn(notify, event)
        await event.wait()
    print('Received notification!')

run(main)
```

Note: Unlike standard library Events, AnyIO events cannot be reused, and must be replaced instead. This practice prevents a class of race conditions, and matches the semantics of the trio library.

1.4.4 Conditions

A condition is basically a combination of an event and a lock. It first acquires a lock and then waits for a notification from the event. Once the condition receives a notification, it releases the lock. The notifying task can also choose to wake up more than one listener at once, or even all of them.

Example:

```
from anyio import create_task_group, create_condition, sleep, run

async def listen(tasknum, condition):
    async with condition:
        await condition.wait()
        print('Woke up task number', tasknum)

async def main():
    condition = create_condition()
    async with create_task_group() as tg:
        for tasknum in range(6):
            await tg.spawn(listen, tasknum, condition)

        await sleep(1)
        async with condition:
            await condition.notify(1)

        await sleep(1)
        async with condition:
            await condition.notify(2)

        await sleep(1)
        async with condition:
            await condition.notify_all()

run(main)
```

1.4.5 Capacity limiters

Capacity limiters are like semaphores except that a single borrower (the current task by default) can only hold a single token at a time. It is also possible to borrow a token on behalf of any arbitrary object, so long as that object is hashable.

Example:

```
from anyio import create_task_group, create_capacity_limiter, sleep, run

async def use_resource(tasknum, limiter):
    async with limiter:
        print('Task number', tasknum, 'is now working with the shared resource')
        await sleep(1)

async def main():
    limiter = create_capacity_limiter(2)
    async with create_task_group() as tg:
        for num in range(10):
```

(continues on next page)

(continued from previous page)

```
await tg.spawn(use_resource, num, limiter)

run(main)
```

To adjust the number of total tokens, you can use the `set_total_tokens()` method.

1.5 Streams

A “stream” in AnyIO is a simple interface for transporting information from one place to another. It can mean either in-process communication or sending data over a network. AnyIO divides streams into two categories: byte streams and object streams.

Byte streams (“Streams” in Trio lingo) are objects that receive and/or send chunks of bytes. They are modelled after the limitations of the stream sockets, meaning the boundaries are not respected. In practice this means that if, for example, you call `.send(b'hello ')` and then `.send(b'world')`, the other end will receive the data chunked in any arbitrary way, like `(b'hello' and b'world')`, `b'hello world'` or `(b'hel', b'lo wo', b'rld')`.

Object streams (“Channels” in Trio lingo), on the other hand, deal with Python objects. The most commonly used implementation of these is the memory object stream. The exact semantics of object streams vary a lot by implementation.

Many stream implementations wrap other streams. Of these, some can wrap any bytes-oriented streams, meaning `ObjectStream[bytes]` and `ByteStream`. This enables many interesting use cases.

1.5.1 Memory object streams

Memory object streams are intended for implementing a producer-consumer pattern with multiple tasks. Using `create_memory_object_stream()`, you get a pair of object streams: one for sending, one for receiving. They essentially work like queues, but with support for closing and asynchronous iteration.

By default, memory object streams are created with a buffer size of 0. This means that `send()` will block until there’s another task that calls `receive()`. You can set the buffer size to a value of your choosing when creating the stream. It is also possible to have an unbounded buffer by passing `:data:math.inf` as the buffer size but this is not recommended.

Memory object streams can be cloned by calling the `clone()` method. Each clone can be closed separately, but each end of the stream is only considered closed once all of its clones have been closed. For example, if you have two clones of the receive stream, the send stream will start raising `BrokenResourceError` only when both receive streams have been closed.

Multiple tasks can send and receive on the same memory object stream (or its clones) but each sent item is only ever delivered to a single recipient.

The receive ends of memory object streams can be iterated using the async iteration protocol. The loop exits when all clones of the send stream have been closed.

Example:

```
from anyio import create_task_group, create_memory_object_stream, run

async def process_items(receive_stream):
    async with receive_stream:
        async for item in receive_stream:
```

(continues on next page)

(continued from previous page)

```

        print('received', item)

    async def main():
        send_stream, receive_stream = create_memory_object_stream()
        async with create_task_group() as tg:
            await tg.spawn(process_items, receive_stream)
            async with send_stream:
                for num in range(10):
                    await send_stream.send(f'number {num}')

run(main)

```

1.5.2 Stapled streams

A stapled stream combines any mutually compatible receive and send stream together, forming a single bidirectional stream.

It comes in two variants:

- *StapledByteStream* (combines a *ByteReceiveStream* with a *ByteSendStream*)
- *StapledObjectStream* (combines an *ObjectReceiveStream* with a compatible *ObjectSendStream*)

1.5.3 Buffered byte streams

A buffered byte stream wraps an existing bytes-oriented receive stream and provides certain amenities that require buffering, such as receiving an exact number of bytes, or receiving until the given delimiter is found.

Example:

```

from anyio import run, create_memory_object_stream
from anyio.streams.buffered import BufferedByteReceiveStream

async def main():
    send, receive = create_memory_object_stream(4)
    buffered = BufferedByteReceiveStream(receive)
    for part in b'hello, world!':
        await send.send(part)

    result = await buffered.receive_exactly(8)
    print(repr(result))

    result = await buffered.receive_until(b'!', 10)
    print(repr(result))

run(main)

```

The above script gives the following output:

```

b'hello, w'
b'orld'

```

1.5.4 Text streams

Text streams wrap existing receive/send streams and encode/decode strings to bytes and vice versa.

Example:

```
from anyio import run, create_memory_object_stream
from anyio.streams.text import TextReceiveStream, TextSendStream

async def main():
    bytes_send, bytes_receive = create_memory_object_stream(1)
    text_send = TextSendStream(bytes_send)
    await text_send.send('ääö')
    result = await bytes_receive.receive()
    print(repr(result))

    text_receive = TextReceiveStream(bytes_receive)
    await bytes_send.send(result)
    result = await text_receive.receive()
    print(repr(result))

run(main)
```

The above script gives the following output:

```
b'\xc3\xa5\xc3\xa4\xc3\xb6'
'ääö'
```

1.5.5 TLS streams

TLS (Transport Layer Security), the successor to SSL (Secure Sockets Layer), is the supported way of providing authenticity and confidentiality for TCP streams in AnyIO.

TLS is typically established right after the connection has been made. The handshake involves the following steps:

- Sending the certificate to the peer (usually just by the server)
- Checking the peer certificate(s) against trusted CA certificates
- Checking that the peer host name matches the certificate

Obtaining a server certificate

There are three principal ways you can get an X.509 certificate for your server:

1. Create a self signed certificate
2. Use [certbot](#) or a similar software to automatically obtain certificates from [Let's Encrypt](#)
3. Buy one from a certificate vendor

The first option is probably the easiest, but this requires that the any client connecting to your server adds the self signed certificate to their list of trusted certificates. This is of course impractical outside of local development and is strongly discouraged in production use.

The second option is nowadays the recommended method, as long as you have an environment where running [certbot](#) or similar software can automatically replace the certificate with a newer one when necessary, and that you don't need any extra features like class 2 validation.

The third option may be your only valid choice when you have special requirements for the certificate that only a certificate vendor can fulfill, or that automatically renewing the certificates is not possible or practical in your environment.

Using self signed certificates

To create a self signed certificate for `localhost`, you can use the `openssl` command line tool:

```
openssl req -x509 -newkey rsa:2048 -subj '/CN=localhost' -keyout key.pem -out cert.
↪pem -nodes -days 365
```

This creates a (2048 bit) private RSA key (`key.pem`) and a certificate (`cert.pem`) matching the host name “localhost”. The certificate will be valid for one year with these settings.

To set up a server using this key-certificate pair:

```
import ssl

from anyio import create_tcp_listener, run
from anyio.streams.tls import TLSListener

async def handle(client):
    async with client:
        name = await client.receive()
        await client.send(b'Hello, %s\n' % name)

async def main():
    # Create a context for the purpose of authenticating clients
    context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)

    # Load the server certificate and private key
    context.load_cert_chain(certfile='cert.pem', keyfile='key.pem')

    # Create the listener and start serving connections
    listener = TLSListener(await create_tcp_listener(local_port=1234), context)
    await listener.serve(handle)

run(main)
```

Connecting to this server can then be done as follows:

```
import ssl

from anyio import connect_tcp, run

async def main():
    # These two steps are only required for certificates that are not trusted by the
    # installed CA certificates on your machine, so you can skip this part if you use
    # Let's Encrypt or a commercial certificate vendor
    context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
    context.load_verify_locations(cafile='cert.pem')

    async with await connect_tcp('localhost', 1234, ssl_context=context) as client:
        await client.send(b'Client\n')
```

(continues on next page)

(continued from previous page)

```
response = await client.receive()
print(response)

run(main)
```

Creating self-signed certificates on the fly

When testing your TLS enabled service, it would be convenient to generate the certificates on the fly. To this end, you can use the `trustme` library:

```
import ssl

import pytest
import trustme

@pytest.fixture(scope='session')
def ca():
    return trustme.CA()

@pytest.fixture(scope='session')
def server_context(ca):
    server_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    ca.issue_cert('localhost').configure_cert(server_context)
    return server_context

@pytest.fixture(scope='session')
def client_context(ca):
    client_context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
    ca.configure_trust(client_context)
    return client_context
```

You can then pass the server and client contexts from the above fixtures to `TLSListener`, `wrap()` or whatever you use on either side.

Dealing with ragged EOFs

According to the [TLS standard](#), encrypted connections should end with a closing handshake. This practice prevents so-called [truncation attacks](#). However, broadly available implementations for protocols such as HTTP, widely ignore this requirement because the protocol level closing signal would make the shutdown handshake redundant.

AnyIO follows the standard by default (unlike the Python standard library's `ssl` module). The practical implication of this is that if you're implementing a protocol that is expected to skip the TLS closing handshake, you need to pass the `standard_compatible=False` option to `wrap()` or `TLSListener`.

1.6 Using typed attributes

On AnyIO, streams and listeners can be layered on top of each other to provide extra functionality. But when you want to look up information from one of the layers down below, you might have to traverse the entire chain to find what you're looking for, which is highly inconvenient. To address this, AnyIO has a system of *typed attributes* where you can look for a specific attribute by its unique key. If a stream or listener wrapper does not have the attribute you're looking for, it will look it up in the wrapped instance, and that wrapper can look in its wrapped instance and so on, until the attribute is either found or the end of the chain is reached. This also lets wrappers override attributes from the wrapped objects when necessary.

A common use case is finding the IP address of the remote side of a TCP connection when the stream may be either *SocketStream* or *TLSSStream*:

```
from anyio import connect_tcp

async def connect(host, port, tls: bool):
    stream = await connect_tcp(host, port, tls=tls)
    print('Connected to', stream.extra(SocketAttribute.remote_address))
```

Each typed attribute provider class should document the set of attributes it provides on its own.

1.6.1 Defining your own typed attributes

By convention, typed attributes are stored together in a container class with other attributes of the same category:

```
from anyio import TypedAttribute, TypedAttributeSet

class MyTypedAttribute:
    string_valued_attribute = TypedAttribute[str]()
    some_float_attribute = TypedAttribute[float]()
```

To provide values for these attributes, implement the *extra_attributes()* property in your class:

```
from anyio import TypedAttributeProvider

class MyAttributeProvider(TypedAttributeProvider):
    def extra_attributes():
        return {
            MyTypedAttribute.string_valued_attribute: lambda: 'my attribute value',
            MyTypedAttribute.some_float_attribute: lambda: 6.492
        }
```

If your class inherits from another typed attribute provider, make sure you include its attributes in the return value:

```
class AnotherAttributeProvider(MyAttributeProvider):
    def extra_attributes():
        return {
            **super().extra_attributes(),
            MyTypedAttribute.string_valued_attribute: lambda: 'overridden attribute_
↪value'
        }
```

1.7 Using sockets and streams

Networking capabilities are arguably the most important part of any asynchronous library. AnyIO contains its own high level implementation of networking on top of low level primitives offered by each of its supported backends.

Currently AnyIO offers the following networking functionality:

- TCP sockets (client + server)
- UNIX domain sockets (client + server)
- UDP sockets

More exotic forms of networking such as raw sockets and SCTP are currently not supported.

Warning: Unlike the standard BSD sockets interface and most other networking libraries, AnyIO (from 2.0 onwards) signals the end of any stream by raising the `EndOfStream` exception instead of returning an empty bytes object.

1.7.1 Working with TCP sockets

TCP (Transmission Control Protocol) is the most commonly used protocol on the Internet. It allows one to connect to a port on a remote host and send and receive data in a reliable manner.

To connect to a listening TCP socket somewhere, you can use `connect_tcp()`:

```
from anyio import connect_tcp, run

async def main():
    async with await connect_tcp('hostname', 1234) as client:
        await client.send(b'Client\n')
        response = await client.receive()
        print(response)

run(main)
```

As a convenience, you can also use `connect_tcp()` to establish a TLS session with the peer after connection, by passing `tls=True` or by passing a nonempty value for either `ssl_context` or `tls_hostname`.

To receive incoming TCP connections, you first create a TCP listener with `create_tcp_listener()` and call `serve()` on it:

```
from anyio import create_tcp_listener, run

async def handle(client):
    async with client:
        name = await client.receive(1024)
        await client.send(b'Hello, %s\n' % name)

async def main():
    listener = await create_tcp_listener(local_port=1234)
    await listener.serve(handle)
```

(continues on next page)

(continued from previous page)

```
run(main)
```

See the section on [TLS streams](#) for more information.

1.7.2 Working with UNIX sockets

UNIX domain sockets are a form of interprocess communication on UNIX-like operating systems. They cannot be used to connect to remote hosts and do not work on Windows.

The API for UNIX domain sockets is much like the one for TCP sockets, except that instead of host/port combinations, you use file system paths.

This is what the client from the TCP example looks like when converted to use UNIX sockets:

```
from anyio import connect_unix, run

async def main():
    async with await connect_unix('/tmp/mysock') as client:
        await client.send(b'Client\n')
        response = await client.receive(1024)
        print(response)

run(main)
```

And the listener:

```
from anyio import create_unix_listener, run

async def handle(client):
    async with client:
        name = await client.receive(1024)
        await client.send(b'Hello, %s\n' % name)

async def main():
    listener = await create_unix_listener('/tmp/mysock')
    await listener.serve(handle)

run(main)
```

1.7.3 Working with UDP sockets

UDP (User Datagram Protocol) is a way of sending packets over the network without features like connections, retries or error correction.

For example, if you wanted to create a UDP “hello” service that just reads a packet and then sends a packet to the sender with the contents prepended with “Hello, “, you would do this:

```
import socket

from anyio import create_udp_socket, run
```

(continues on next page)

(continued from previous page)

```
async def main():
    async with await create_udp_socket(family=socket.AF_INET, local_port=1234) as udp:
        async for packet, (host, port) in udp:
            await udp.sendto(b'Hello, ' + packet, host, port)

run(main)
```

Note: If you are testing on your local machine or don't know which family socket to use, it is a good idea to replace `family=socket.AF_INET` by `local_host='localhost'` in the previous example.

If your use case involves sending lots of packets to a single destination, you can still “connect” your UDP socket to a specific host and port to avoid having to pass the address and port every time you send data to the peer:

```
from anyio import create_connected_udp_socket, run

async def main():
    async with await create_connected_udp_socket(
        remote_host='hostname', remote_port=1234) as udp:
        await udp.send(b'Hi there!\n')

run(main)
```

1.8 Working with threads

Practical asynchronous applications occasionally need to run network, file or computationally expensive operations. Such operations would normally block the asynchronous event loop, leading to performance issues. The solution is to run such code in *worker threads*. Using worker threads lets the event loop continue running other tasks while the worker thread runs the blocking call.

Caution: Do not spawn too many threads, as the context switching overhead may cause your system to slow down to a crawl. A few dozen threads should be fine, but hundreds are probably bad. Consider using AnyIO's semaphores to limit the maximum number of threads.

1.8.1 Running a function in a worker thread

To run a (synchronous) callable in a worker thread:

```
import time

from anyio import run_sync_in_worker_thread, run

async def main():
    await run_sync_in_worker_thread(time.sleep, 5)

run(main)
```

By default, tasks are shielded from cancellation while they are waiting for a worker thread to finish. You can pass the `cancellable=True` parameter to allow such tasks to be cancelled. Note, however, that the thread will still continue running – only its outcome will be ignored.

1.8.2 Calling asynchronous code from a worker thread

If you need to call a coroutine function from a worker thread, you can do this:

```
from anyio import run_async_from_thread, sleep, run_sync_in_worker_thread, run

def blocking_function():
    run_async_from_thread(sleep, 5)

async def main():
    await run_sync_in_worker_thread(blocking_function)

run(main)
```

Note: The worker thread must have been spawned using `run_sync_in_worker_thread()` for this to work.

1.8.3 Calling asynchronous code from an external thread

If you need to run async code from a thread that is not a worker thread spawned by the event loop, you need a *blocking portal*. This needs to be obtained from within the event loop thread.

One way to do this is to start a new event loop with a portal, using `start_blocking_portal()` (which takes mostly the same arguments as `run()`):

```
from anyio import start_blocking_portal

portal = start_blocking_portal(backend='trio')
portal.call(...)

# At the end of your application, stop the portal
portal.stop_from_external_thread()
```

Or, you can use it as a context manager if that suits your use case:

```
with start_blocking_portal(backend='trio') as portal:
    portal.call(...)
```

If you already have an event loop running and wish to grant access to external threads, you can use `create_blocking_portal()` directly:

```
from anyio import create_blocking_portal, run

async def main():
    async with create_blocking_portal() as portal:
        # ...hand off the portal to external threads...
```

(continues on next page)

(continued from previous page)

```
        await portal.sleep_until_stopped()

anyio.run(main)
```

1.8.4 Spawning tasks from worker threads

When you need to spawn a task to be run in the background, you can do so using `spawn_task()`:

```
from concurrent.futures import as_completed

from anyio import start_blocking_portal, sleep


async def long_running_task(index):
    await sleep(1)
    print(f'Task {index} running...')
    await sleep(index)
    return f'Task {index} return value'


with start_blocking_portal() as portal:
    futures = [portal.spawn_task(long_running_task, i) for i in range(1, 5)]
    for future in as_completed(futures):
        print(future.result())
```

Cancelling tasks spawned this way can be done by cancelling the returned `Future`.

1.8.5 Using asynchronous context managers from worker threads

You can use `wrap_async_context_manager()` to wrap an asynchronous context managers as a synchronous one:

```
from anyio import start_blocking_portal


class AsyncContextManager:
    async def __aenter__(self):
        print('entering')

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print('exiting with', exc_type)


async_cm = AsyncContextManager()
with start_blocking_portal() as portal, portal.wrap_async_context_manager(async_cm):
    print('inside the context manager block')
```

Note: You cannot use wrapped async context managers in synchronous callbacks inside the event loop thread.

Note: The `__aenter__()` and `__aexit__()` methods will be called from different tasks so a task group as the

async context manager will not work here.

1.9 Using subprocesses

AnyIO allows you to run arbitrary executables in subprocesses, either as a one-shot call or by opening a process handle for you that gives you more control over the subprocess.

You can either give the command as a string, in which case it is passed to your default shell (equivalent to `shell=True` in `subprocess.run()`), or as a sequence of strings (`shell=False`) in which the executable is the first item in the sequence and the rest are arguments passed to it.

Note: The subprocess facilities provided by AnyIO do not include a way to execute arbitrary Python code like the `multiprocessing` module does, but they can be used as building blocks for such a feature.

1.9.1 Running one-shot commands

To run an external command with one call, use `run_process()`:

```
from anyio import run_process, run

async def main():
    result = await run_process('ps')
    print(result.stdout.decode())

run(main)
```

The snippet above runs the `ps` command within a shell (. To run it directly:

```
from anyio import run_process, run

async def main():
    result = await run_process(['ps'])
    print(result.stdout.decode())

run(main)
```

1.9.2 Working with processes

When you have more complex requirements for your interaction with subprocesses, you can launch one with `open_process()`:

```
from anyio import open_process, run
from anyio.streams.text import TextReceiveStream

async def main():
    async with await open_process(['ps']) as process:
        async for text in TextReceiveStream(process.stdout):
```

(continues on next page)

(continued from previous page)

```
        print(text)

run(main)
```

See the API documentation of *Process* for more information.

1.10 Asynchronous file I/O support

AnyIO provides asynchronous wrappers for blocking file operations. These wrappers run blocking operations in worker threads.

Example:

```
from anyio import open_file, run

async def main():
    async with await open_file('/some/path/somewhere') as f:
        contents = await f.read()
        print(contents)

run(main)
```

The wrappers also support asynchronous iteration of the file line by line, just as the standard file objects support synchronous iteration:

```
from anyio import open_file, run

async def main():
    async with await open_file('/some/path/somewhere') as f:
        async for line in f:
            print(line, end='')

run(main)
```

1.11 Receiving operating system signals

You may occasionally find it useful to receive signals sent to your application in a meaningful way. For example, when you receive a `signal.SIGTERM` signal, your application is expected to shut down gracefully. Likewise, `SIGHUP` is often used as a means to ask the application to reload its configuration.

AnyIO provides a simple mechanism for you to receive the signals you're interested in:

```
import signal

from anyio import open_signal_receiver, run

async def main():
    async with open_signal_receiver(signal.SIGTERM, signal.SIGHUP) as signals:
        async for signum in signals:
```

(continues on next page)

(continued from previous page)

```

    if signum == signal.SIGTERM:
        return
    elif signum == signal.SIGHUP:
        print('Reloading configuration')

run(main)

```

Note: Signal handlers can only be installed in the main thread, so they will not work when the event loop is being run through `start_blocking_portal()`, for instance.

Note: Windows does not natively support signals so do not rely on this in a cross platform application.

1.11.1 Handling KeyboardInterrupt and SystemExit

By default, different backends handle the Ctrl+C (or Ctrl+Break on Windows) key combination and external termination (`KeyboardInterrupt` and `SystemExit`, respectively) differently: trio raises the relevant exception inside the application while asyncio and curio shut down all the tasks and exit. If you need to do your own cleanup in these situations, you will need to install a signal handler:

```

import signal

from anyio import open_signal_receiver, create_task_group, run
from anyio.abc import CancelScope

async def signal_handler(scope: CancelScope):
    async with open_signal_receiver(signal.SIGINT, signal.SIGTERM) as signals:
        async for signum in signals:
            if signum == signal.SIGINT:
                print('Ctrl+C pressed!')
            else:
                print('Terminated!')

            await scope.cancel()
        return

async def main():
    async with create_task_group() as tg:
        await tg.spawn(signal_handler, tg.cancel_scope)
        ... # proceed with starting the actual application logic

run(main)

```

Note: Windows does not support the `SIGTERM` signal so if you need a mechanism for graceful shutdown on Windows, you will have to find another way.

1.12 Testing with AnyIO

AnyIO provides built-in support for testing your library or application in the form of a `pytest` plugin.

1.12.1 Creating asynchronous tests

Pytest does not natively support running asynchronous test functions, so they have to be marked for the AnyIO pytest plugin to pick them up. This can be done in one of two ways:

1. Using the `pytest.mark.anyio` marker
2. Using the `anyio_backend` fixture, either directly or via another fixture

The simplest way is thus the following:

```
import pytest

# This is the same as using the @pytest.mark.anyio on all test functions in the module
pytestmark = pytest.mark.anyio

async def test_something():
    ...
```

Marking modules, classes or functions with this marker has the same effect as applying the `pytest.mark.usefixtures('anyio_backend')` on them.

Thus, you can also require the fixture directly in your tests and fixtures:

```
import pytest

async def test_something(anyio_backend):
    ...
```

1.12.2 Specifying the backends to run on

The `anyio_backend` fixture determines the backends and their options that tests and fixtures are run with. The AnyIO pytest plugin comes with a function scoped fixture with this name which runs everything on all supported backends.

If you change the backends/options for the entire project, then put something like this in your top level `conftest.py`:

```
@pytest.fixture
def anyio_backend():
    return 'asyncio'
```

If you want to specify different options for the selected backend, you can do so by passing a tuple of (backend name, options dict):

```
@pytest.fixture(params=[
    pytest.param(('asyncio', {'use_uvloop': True}), id='asyncio+uvloop'),
    pytest.param(('asyncio', {'use_uvloop': False}), id='asyncio'),
    pytest.param('curio'),
```

(continues on next page)

(continued from previous page)

```

    pytest.param(('trio', {'restrict_keyboard_interrupt_to_checkpoints': True}), id=
    ↪ 'trio')
])
def anyio_backend(request):
    return request.param

```

If you need to run a single test on a specific backend, you can use `@pytest.mark.parametrize` (remember to add the `anyio_backend` parameter to the actual test function, or pytest will complain):

```

@pytest.mark.parametrize('anyio_backend', ['asyncio'])
async def test_on_asyncio_only(anyio_backend):
    ...

```

Because the `anyio_backend` fixture can return either a string or a tuple, there are two additional function-scoped fixtures (which themselves depend on the `anyio_backend` fixture) provided for your convenience:

- `anyio_backend_name`: the name of the backend (e.g. `asyncio`)
- `anyio_backend_options`: the dictionary of option keywords used to run the backend

1.12.3 Asynchronous fixtures

The plugin also supports coroutine functions as fixtures, for the purpose of setting up and tearing down asynchronous services used for tests.

There are two ways to get the AnyIO pytest plugin to run your asynchronous fixtures:

1. Use them in AnyIO enabled tests (see the first section)
2. Use the `anyio_backend` fixture (or any other fixture using it) in the fixture itself

The simplest way is using the first option:

```

import pytest

pytestmark = pytest.mark.anyio

@pytest.fixture
async def server():
    server = await setup_server()
    yield server
    await server.shutdown()

async def test_server(server):
    result = await server.do_something()
    assert result == 'foo'

```

For `autouse=True` fixtures, you may need to use the other approach:

```

@pytest.fixture(autouse=True)
async def server(anyio_backend):
    server = await setup_server()
    yield
    await server.shutdown()

```

(continues on next page)

(continued from previous page)

```
async def test_server():
    result = await client.do_something_on_the_server()
    assert result == 'foo'
```

1.12.4 Using async fixtures with higher scopes

For async fixtures with scopes other than function, you will need to define your own `anyio_backend` fixture because the default `anyio_backend` fixture is function scoped:

```
@pytest.fixture(scope='module')
def anyio_backend():
    return 'asyncio'

@pytest.fixture(scope='module')
async def server(anyio_backend):
    server = await setup_server()
    yield
    await server.shutdown()
```

1.12.5 Technical details

The fixtures and tests are run by a “test runner”, implemented separately for each backend. The test runner keeps an event loop open during the request, making it possible for code in fixtures to communicate with the code in the tests (and each other).

The test runner is created when the first matching async test or fixture is about to be run, and shut down when that same fixture is being torn down or the test has finished running. As such, if no async fixtures are used, a separate test runner is created for each test. Conversely, if even one async fixture (scoped higher than `function`) is shared across all tests, only one test runner will be created during the test session.

1.13 API reference

1.13.1 Event loop

`anyio.run(func, *args, backend='asyncio', backend_options=None)`

Run the given coroutine function in an asynchronous event loop.

The current thread must not be already running an event loop.

Parameters

- **func** (`Callable[... Coroutine[Any, Any, +T_Retval]]`) – a coroutine function
- **args** – positional arguments to `func`
- **backend** (`str`) – name of the asynchronous event loop implementation – one of `asyncio`, `curio` and `trio`
- **backend_options** (`Optional[Dict[str, Any]]`) – keyword arguments to call the `backend.run()` implementation with (documented [here](#))

Return type `+T_Return`

Returns the return value of the coroutine function

Raises

- **RuntimeError** – if an asynchronous event loop is already running in this thread
- **LookupError** – if the named backend is not found

`anyio.get_all_backends()`

Return a tuple of the names of all built-in backends.

Return type `Tuple[str, ...]`

`anyio.get_cancelled_exc_class()`

Return the current async library's cancellation exception class.

Return type `Type[BaseException]`

async `anyio.sleep(delay)`

Pause the current task for the specified duration.

Parameters `delay (float)` – the duration, in seconds

Return type `None`

async `anyio.current_time()`

Return the current value of the event loop's internal clock.

Return type `float`

Returns the clock value (seconds)

1.13.2 Asynchronous resources

async `anyio.aclose_forcefully(resource)`

Close an asynchronous resource in a cancelled scope.

Doing this closes the resource without waiting on anything.

Parameters `resource (AsyncResource)` – the resource to close

Return type `None`

class `anyio.abc.AsyncResource`

Bases: `object`

Abstract base class for all closeable asynchronous resources.

Works as an asynchronous context manager which returns the instance itself on enter, and calls `aclose()` on exit.

abstract async `aclose()`

Close the resource.

Return type `None`

1.13.3 Typed attributes

`anyio.typing_attribute()`

Return a unique object, used to mark typed attributes.

class `anyio.TypedAttributeSet`

Bases: `object`

Superclass for typed attribute collections.

Checks that every public attribute of every subclass has a type annotation.

class `anyio.TypedAttributeProvider`

Bases: `object`

Base class for classes that wish to provide typed extra attributes.

extra (*attribute*, *default=undefined*)

Return the value of the given typed extra attribute.

Parameters

- **attribute** – the attribute (member of a `TypedAttributeSet`) to look for
- **default** – the value that should be returned if no value is found for the attribute

Raises `TypedAttributeLookupError` – if the search failed and no default value was given

property `extra_attributes`

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

Return type `Mapping[~T_Attr, Callable[[], ~T_Attr]]`

1.13.4 Timeouts and cancellation

`anyio.open_cancel_scope(*, shield=False)`

Open a cancel scope.

Parameters **shield** (`bool`) – True to shield the cancel scope from external cancellation

Return type `CancelScope`

Returns a cancel scope

`anyio.move_on_after(delay, *, shield=False)`

Create an async context manager which is exited if it does not complete within the given time.

Parameters

- **delay** (`Optional[float]`) – maximum allowed time (in seconds) before exiting the context block, or `None` to disable the timeout
- **shield** (`bool`) – True to shield the cancel scope from external cancellation

Returns an asynchronous context manager that yields a cancel scope

Return type `AsyncContextManager[CancelScope]`

`anyio.fail_after(delay, *, shield=False)`

Create an async context manager which raises an exception if does not finish in time.

Parameters

- **delay** (`Optional[float]`) – maximum allowed time (in seconds) before raising the exception, or `None` to disable the timeout
- **shield** (`bool`) – True to shield the cancel scope from external cancellation

Returns an asynchronous context manager that yields a cancel scope

Return type `AsyncContextManager[CancelScope]`

Raises `TimeoutError` – if the block does not complete within the allotted time

`anyio.current_effective_deadline()`

Return the nearest deadline among all the cancel scopes effective for the current task.

Returns a clock value from the event loop's internal clock (`float('inf')` if there is no deadline in effect)

Return type `float`

class `anyio.abc.CancelScope`

Bases: `object`

abstract async cancel()

Cancel this scope immediately.

Return type `None`

abstract property cancel_called

True if `cancel()` has been called.

Return type `bool`

abstract property deadline

The time (clock value) when this scope is cancelled automatically.

Will be `float('inf')` if no timeout has been set.

Return type `float`

abstract property shield

True if this scope is shielded from external cancellation.

While a scope is shielded, it will not receive cancellations from outside.

Return type `bool`

1.13.5 Task groups

`anyio.create_task_group()`

Create a task group.

Return type `TaskGroup`

Returns a task group

class `anyio.abc.TaskGroup`

Bases: `object`

Groups several asynchronous tasks together.

Variables `cancel_scope` (`CancelScope`) – the cancel scope inherited by all child tasks

abstract async spawn (`func, *args, name=None`)

Launch a new task in this task group.

Parameters

- **func** (`Callable[... Coroutine]`) – a coroutine function
- **args** – positional arguments to call the function with
- **name** – name of the task, for the purposes of introspection and debugging

Return type `None`

1.13.6 Threads

async `anyio.run_sync_in_worker_thread(func, *args, cancellable=False, limiter=None)`

Call the given function with the given arguments in a worker thread.

If the `cancellable` option is enabled and the task waiting for its completion is cancelled, the thread will still run its course but its return value (or any raised exception) will be ignored.

Parameters

- **func** (`Callable[... ~T_Retval]`) – a callable
- **args** – positional arguments for the callable
- **cancellable** (`bool`) – True to allow cancellation of the operation
- **limiter** (`Optional[CapacityLimiter]`) – capacity limiter to use to limit the total amount of threads running (if omitted, the default limiter is used)

Return type `~T_Retval`**Returns** an awaitable that yields the return value of the function.**anyio.run_async_from_thread(func, *args)**

Call a coroutine function from a worker thread.

Parameters

- **func** (`Callable[... Coroutine[Any, Any, ~T_Retval]]`) – a coroutine function
- **args** – positional arguments for the callable

Return type `~T_Retval`**Returns** the return value of the coroutine function**anyio.current_default_worker_thread_limiter()**

Return the capacity limiter that is used by default to limit the number of concurrent threads.

Return type `CapacityLimiter`**Returns** a capacity limiter object**anyio.create_blocking_portal()**

Create a portal for running functions in the event loop thread from external threads.

Use this function in asynchronous code when you need to allow external threads access to the event loop where your asynchronous code is currently running.

Return type `BlockingPortal`**anyio.start_blocking_portal(backend='asyncio', backend_options=None)**

Start a new event loop in a new thread and run a blocking portal in its main task.

The parameters are the same as for `run()`.**Parameters**

- **backend** (*str*) – name of the backend
- **backend_options** (*Optional[Dict[str, Any]]*) – backend options

Return type *BlockingPortal*

Returns a blocking portal object

class `anyio.abc.BlockingPortal`

Bases: `object`

An object tied that lets external threads run code in an asynchronous event loop.

call (*func, *args*)

Call the given function in the event loop thread.

If the callable returns a coroutine object, it is awaited on.

Parameters **func** – any callable

Raises `RuntimeError` – if the portal is not running or if this method is called from within the event loop thread

async sleep_until_stopped ()

Sleep until `stop()` is called.

Return type `None`

spawn_task (*func, *args*)

Spawn a task in the portal's task group.

The task will be run inside a cancel scope which can be cancelled by cancelling the returned future.

Parameters

- **func** (*Callable[... Coroutine]*) – the target coroutine function
- **args** – positional arguments passed to `func`

Return type `Future`

Returns a future that resolves with the return value of the callable if the task completes successfully, or with the exception raised in the task

Raises `RuntimeError` – if the portal is not running or if this method is called from within the event loop thread

New in version 2.1.

async stop (*cancel_remaining=False*)

Signal the portal to shut down.

This marks the portal as no longer accepting new calls and exits from `sleep_until_stopped()`.

Parameters **cancel_remaining** (*bool*) – True to cancel all the remaining tasks, False to let them finish before returning

Return type `None`

stop_from_external_thread (*cancel_remaining=False*)

Signal the portal to stop and wait for the event loop thread to finish.

Parameters **cancel_remaining** (*bool*) – True to cancel all the remaining tasks, False to let them finish before returning

Return type `None`

wrap_async_context_manager (*cm*)

Wrap an async context manager as a synchronous context manager via this portal.

Spawns a task that will call both `__aenter__()` and `__aexit__()`, stopping in the middle until the synchronous context manager exits.

Parameters *cm* (`AsyncContextManager[+T_co]`) – an asynchronous context manager

Return type `ContextManager[+T_co]`

Returns a synchronous context manager

New in version 2.1.

1.13.7 Async file I/O

async `anyio.open_file` (*file*, *mode*='r', *buffering*=-1, *encoding*=None, *errors*=None, *newline*=None, *closefd*=True, *opener*=None)

Open a file asynchronously.

The arguments are exactly the same as for the builtin `open()`.

Return type `AsyncFile`

Returns an asynchronous file object

class `anyio.AsyncFile` (*fp*)

Bases: `anyio.abc.AsyncResource`

An asynchronous file object.

This class wraps a standard file object and provides async friendly versions of the following blocking methods (where available on the original file object):

- `read`
- `read1`
- `readline`
- `readlines`
- `readinto`
- `readinto1`
- `write`
- `writelines`
- `truncate`
- `seek`
- `tell`
- `flush`

All other methods are directly passed through.

This class supports the asynchronous context manager protocol which closes the underlying file at the end of the context block.

This class also supports asynchronous iteration:

```

async with await anyio.open_file(...) as f:
    async for line in f:
        print(line)

```

async aclose()

Close the resource.

Return type `None`

property wrapped

The wrapped file object.

1.13.8 Streams and stream wrappers

`anyio.create_memory_object_stream(max_buffer_size=0, item_type=None)`

Create a memory object stream.

Parameters

- **max_buffer_size** – number of items held in the buffer until `send()` starts blocking
- **item_type** – type of item, for marking the streams with the right generic type for static typing (not used at run time)

Returns a tuple of (send stream, receive stream)

class `anyio.abc.UnreliableObjectReceiveStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.AsyncResource`, `anyio.TypedAttributeProvider`

An interface for receiving objects.

This interface makes no guarantees that the received messages arrive in the order in which they were sent, or that no messages are missed.

Asynchronously iterating over objects of this type will yield objects matching the given type parameter.

abstract async receive()

Receive the next item.

Raises

- `ClosedResourceError` – if the receive stream has been explicitly closed
- `EndOfStream` – if this stream has been closed from the other end
- `BrokenResourceError` – if this stream has been rendered unusable due to external causes

Return type `~T_Item`

class `anyio.abc.UnreliableObjectSendStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.AsyncResource`, `anyio.TypedAttributeProvider`

An interface for sending objects.

This interface makes no guarantees that the messages sent will reach the recipient(s) in the same order in which they were sent, or at all.

abstract async send(item)

Send an item to the peer(s).

Parameters `item (~T_Item)` – the item to send

Raises

- `ClosedResourceError` – if the send stream has been explicitly closed
- `BrokenResourceError` – if this stream has been rendered unusable due to external causes

Return type `None`

class `anyio.abc.UnreliableObjectStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.streams.UnreliableObjectReceiveStream[typing.T_Item]`, `anyio.abc.streams.UnreliableObjectSendStream[typing.T_Item]`

A bidirectional message stream which does not guarantee the order or reliability of message delivery.

class `anyio.abc.ObjectReceiveStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.streams.UnreliableObjectReceiveStream[typing.T_Item]`

A receive message stream which guarantees that messages are received in the same order in which they were sent, and that no messages are missed.

class `anyio.abc.ObjectSendStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.streams.UnreliableObjectSendStream[typing.T_Item]`

A send message stream which guarantees that messages are delivered in the same order in which they were sent, without missing any messages in the middle.

class `anyio.abc.ObjectStream`

Bases: `Generic[typing.T_Item]`, `anyio.abc.streams.ObjectReceiveStream[typing.T_Item]`, `anyio.abc.streams.ObjectSendStream[typing.T_Item]`, `anyio.abc.streams.UnreliableObjectStream[typing.T_Item]`

A bidirectional message stream which guarantees the order and reliability of message delivery.

abstract `async send_eof()`

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type `None`

class `anyio.abc.ByteReceiveStream`

Bases: `anyio.abc.AsyncResource`, `anyio.TypedAttributeProvider`

An interface for receiving bytes from a single peer.

Iterating this byte stream will yield a byte string of arbitrary length, but no more than 65536 bytes.

abstract `async receive (max_bytes=65536)`

Receive at most `max_bytes` bytes from the peer.

Parameters `max_bytes (int)` – maximum number of bytes to receive

Return type `bytes`

Returns the received bytes

Raises `EndOfStream` – if this stream has been closed from the other end

class `anyio.abc.ByteSendStream`

Bases: `anyio.abc.AsyncResource`, `anyio.TypedAttributeProvider`

An interface for sending bytes to a single peer.

abstract `async send(item)`

Send the given bytes to the peer.

Parameters `item` (`bytes`) – the bytes to send

Return type `None`

class `anyio.abc.ByteStream`

Bases: `anyio.abc.ByteReceiveStream`, `anyio.abc.ByteSendStream`

A bidirectional byte stream.

abstract `async send_eof()`

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type `None`

class `anyio.abc.Listener(*args, **kwargs)`

Bases: `Generic[typing.T_Stream]`, `anyio.abc.AsyncResource`, `anyio.TypedAttributeProvider`

An interface for objects that let you accept incoming connections.

abstract `async serve(handler, task_group=None)`

Accept incoming connections as they come in and spawn tasks to handle them.

Parameters

- **handler** (`Callable[[~T_Stream], Any]`) – a callable that will be used to handle each accepted connection
- **task_group** (`Optional[TaskGroup]`) – the task group that will be used to spawn tasks for handling each accepted connection (if omitted, an ad-hoc task group will be created)

Return type `None`

`anyio.abc.AnyUnreliableByteReceiveStream = typing.Union[anyio.abc.UnreliableObjectReceiveStream`

`Union type; Union[X, Y] means either X or Y.`

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- `None` as an argument is a special case and is replaced by `type(None)`.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].

```
anyio.abc.AnyUnreliableByteSendStream = typing.Union[anyio.abc.UnreliableObjectSendStream[...],
Union type; Union[X, Y] means either X or Y.
```

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.

- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

`anyio.abc.AnyUnreliableByteStream = typing.Union[anyio.abc.UnreliableObjectStream[bytes], anyio.abc.UnreliableByteStream]`
 Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- `None` as an argument is a special case and is replaced by `type(None)`.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

`anyio.abc.AnyByteReceiveStream = typing.Union[anyio.abc.ObjectReceiveStream[bytes], anyio.abc.AnyUnreliableByteStream]`
 Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- `None` as an argument is a special case and is replaced by `type(None)`.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].

```
anyio.abc.AnyByteSendStream = typing.Union[anyio.abc.ObjectSendStream[bytes], anyio.abc.ByteStream]
Union type; Union[X, Y] means either X or Y.
```

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.

- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

`anyio.abc.AnyByteStream = typing.Union[anyio.abc.ObjectStream[bytes], anyio.abc.ByteStream]`
 Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- `None` as an argument is a special case and is replaced by `type(None)`.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

class `anyio.streams.buffered.BufferedByteReceiveStream(receive_stream)`

Bases: `anyio.abc.ByteReceiveStream`

Wraps any bytes-based receive stream and uses a buffer to provide sophisticated receiving capabilities in the form of a byte stream.

async `aclose()`

Close the resource.

Return type `None`

property `buffer`

The bytes currently in the buffer.

Return type `bytes`

property `extra_attributes`

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async receive (*max_bytes=65536*)

Receive at most *max_bytes* bytes from the peer.

Parameters *max_bytes* (*int*) – maximum number of bytes to receive

Return type *bytes*

Returns the received bytes

Raises *EndOfStream* – if this stream has been closed from the other end

async receive_exactly (*nbytes*)

Read exactly the given amount of bytes from the stream.

Parameters *nbytes* (*int*) – the number of bytes to read

Return type *bytes*

Returns the bytes read

Raises *IncompleteRead* – if the stream was closed before the requested amount of bytes could be read from the stream

async receive_until (*delimiter, max_bytes*)

Read from the stream until the delimiter is found or *max_bytes* have been read.

Parameters

- *delimiter* (*bytes*) – the marker to look for in the stream
- *max_bytes* (*int*) – maximum number of bytes that will be read before raising *DelimiterNotFound*

Return type *bytes*

Returns the bytes read (not including the delimiter)

Raises

- *IncompleteRead* – if the stream was closed before the delimiter was found
- *DelimiterNotFound* – if the delimiter is not found within the bytes read up to the maximum allowed

class *anyio.streams.memory.MemoryObjectReceiveStream* (*_state*)

Bases: *Generic*[*typing.T_Item*], *anyio.abc.ObjectReceiveStream*[*typing.T_Item*]

async aclose ()

Close the resource.

Return type *None*

clone ()

Create a clone of this receive stream.

Each clone can be closed separately. Only when all clones have been closed will the receiving end of the memory stream be considered closed by the sending ends.

Return type *MemoryObjectReceiveStream*

Returns the cloned stream

async receive ()

Receive the next item.

Raises

- *ClosedResourceError* – if the receive stream has been explicitly closed
- *EndOfStream* – if this stream has been closed from the other end
- *BrokenResourceError* – if this stream has been rendered unusable due to external causes

Return type *~T_Item*

async receive_nowait()

Receive the next item if it can be done without waiting.

Return type *~T_Item*

Returns the received item

Raises

- *ClosedResourceError* – if this send stream has been closed
- *EndOfStream* – if the buffer is empty and this stream has been closed from the sending end
- *WouldBlock* – if there are no items in the buffer and no tasks waiting to send

class `anyio.streams.memory.MemoryObjectSendStream(_state)`

Bases: `Generic[typing.T_Item]`, `anyio.abc.ObjectSendStream[typing.T_Item]`

async aclose()

Close the resource.

Return type *None*

clone()

Create a clone of this send stream.

Each clone can be closed separately. Only when all clones have been closed will the sending end of the memory stream be considered closed by the receiving ends.

Return type *MemoryObjectSendStream*

Returns the cloned stream

async send(item)

Send an item to the peer(s).

Parameters *item* (*~T_Item*) – the item to send

Raises

- *ClosedResourceError* – if the send stream has been explicitly closed
- *BrokenResourceError* – if this stream has been rendered unusable due to external causes

Return type *None*

async send_nowait(item)

Send an item immediately if it can be done without waiting.

Parameters *item* (*~T_Item*) – the item to send

Raises

- *ClosedResourceError* – if this send stream has been closed

- **BrokenResourceError** – if the stream has been closed from the receiving end
- **WouldBlock** – if the buffer is full and there are no tasks waiting to receive

Return type `None`

class `anyio.streams.stapled.MultiListener` (*listeners*)

Bases: `Generic[typing.T_Stream]`, `anyio.abc.Listener[typing.T_Stream]`

Combines multiple listeners into one, serving connections from all of them at once.

Any MultiListeners in the given collection of listeners will have their listeners moved into this one.

Extra attributes are provided from each listener, with each successive listener overriding any conflicting attributes from the previous one.

Parameters `listeners` (`Sequence[Listener[T_Stream]]`) – listeners to serve

async `aclose()`

Close the resource.

Return type `None`

property `extra_attributes`

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async `serve(handler, task_group=None)`

Accept incoming connections as they come in and spawn tasks to handle them.

Parameters

- **handler** (`Callable[[~T_Stream], Any]`) – a callable that will be used to handle each accepted connection
- **task_group** (`Optional[TaskGroup]`) – the task group that will be used to spawn tasks for handling each accepted connection (if omitted, an ad-hoc task group will be created)

Return type `None`

class `anyio.streams.stapled.StapledByteStream` (*send_stream, receive_stream*)

Bases: `anyio.abc.ByteStream`

Combines two byte streams into a single, bidirectional byte stream.

Extra attributes will be provided from both streams, with the receive stream providing the values in case of a conflict.

Parameters

- **send_stream** (`ByteSendStream`) – the sending byte stream
- **receive_stream** (`ByteReceiveStream`) – the receiving byte stream

async `aclose()`

Close the resource.

Return type `None`

property `extra_attributes`

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async receive (*max_bytes=65536*)

Receive at most *max_bytes* bytes from the peer.

Parameters *max_bytes* (*int*) – maximum number of bytes to receive

Return type *bytes*

Returns the received bytes

Raises *EndOfStream* – if this stream has been closed from the other end

async send (*item*)

Send the given bytes to the peer.

Parameters *item* (*bytes*) – the bytes to send

Return type *None*

async send_eof ()

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type *None*

class *anyio.streams.stapled.StapledObjectStream* (*send_stream, receive_stream*)

Bases: *Generic*[*typing.T_Item*], *anyio.abc.ObjectStream*[*typing.T_Item*]

Combines two object streams into a single, bidirectional object stream.

Extra attributes will be provided from both streams, with the receive stream providing the values in case of a conflict.

Parameters

- *send_stream* (*ObjectSendStream*) – the sending object stream
- *receive_stream* (*ObjectReceiveStream*) – the receiving object stream

async aclose ()

Close the resource.

Return type *None*

property *extra_attributes*

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async receive ()

Receive the next item.

Raises

- *ClosedResourceError* – if the receive stream has been explicitly closed
- *EndOfStream* – if this stream has been closed from the other end
- *BrokenResourceError* – if this stream has been rendered unusable due to external causes

Return type *~T_Item*

async send (*item*)

Send an item to the peer(s).

Parameters `item (~T_Item)` – the item to send

Raises

- [`ClosedResourceError`](#) – if the send stream has been explicitly closed
- [`BrokenResourceError`](#) – if this stream has been rendered unusable due to external causes

Return type `None`

async send_eof()

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type `None`

class `anyio.streams.text.TextReceiveStream`(*transport_stream*, *encoding='utf-8'*, *errors='strict'*)

Bases: `anyio.abc.ObjectReceiveStream[str]`

Stream wrapper that decodes bytes to strings using the given encoding.

Decoding is done using [`IncrementalDecoder`](#) which returns any completely received unicode characters as soon as they come in.

Parameters

- **transport_stream** (`Union[ObjectReceiveStream[bytes], ByteReceiveStream]`) – any bytes-based receive stream
- **encoding** (`InitVar`) – character encoding to use for decoding bytes to strings (defaults to `utf-8`)
- **errors** (`InitVar`) – handling scheme for decoding errors (defaults to `strict`; see the [codecs module documentation](#) for a comprehensive list of options)

async aclose()

Close the resource.

Return type `None`

property extra_attributes

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async receive()

Receive the next item.

Raises

- [`ClosedResourceError`](#) – if the receive stream has been explicitly closed
- [`EndOfStream`](#) – if this stream has been closed from the other end
- [`BrokenResourceError`](#) – if this stream has been rendered unusable due to external causes

Return type `str`

class `anyio.streams.text.TextSendStream`(*transport_stream*, *encoding='utf-8'*, *errors='strict'*)

Bases: `anyio.abc.ObjectSendStream[str]`

Sends strings to the wrapped stream as bytes using the given encoding.

Parameters

- **transport_stream** (*AnyByteSendStream*) – any bytes-based send stream
- **encoding** (*str*) – character encoding to use for encoding strings to bytes (defaults to `utf-8`)
- **errors** (*str*) – handling scheme for encoding errors (defaults to `strict`; see the [codecs module documentation](#) for a comprehensive list of options)

async **aclose** ()

Close the resource.

Return type `None`

property **extra_attributes**

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async **send** (*item*)

Send an item to the peer(s).

Parameters **item** (*str*) – the item to send

Raises

- **ClosedResourceError** – if the send stream has been explicitly closed
- **BrokenResourceError** – if this stream has been rendered unusable due to external causes

Return type `None`

class `anyio.streams.text.TextStream` (*transport_stream*, *encoding='utf-8'*, *errors='strict'*)

Bases: `anyio.abc.ObjectStream`[*str*]

A bidirectional stream that decodes bytes to strings on receive and encodes strings to bytes on send.

Extra attributes will be provided from both streams, with the receive stream providing the values in case of a conflict.

Parameters

- **transport_stream** (*AnyByteStream*) – any bytes-based stream
- **encoding** (*str*) – character encoding to use for encoding/decoding strings to/from bytes (defaults to `utf-8`)
- **errors** (*str*) – handling scheme for encoding errors (defaults to `strict`; see the [codecs module documentation](#) for a comprehensive list of options)

async **aclose** ()

Close the resource.

Return type `None`

property **extra_attributes**

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async receive()

Receive the next item.

Raises

- ***ClosedResourceError*** – if the receive stream has been explicitly closed
- ***EndOfStream*** – if this stream has been closed from the other end
- ***BrokenResourceError*** – if this stream has been rendered unusable due to external causes

Return type `str`

async send(item)

Send an item to the peer(s).

Parameters **item** (`str`) – the item to send

Raises

- ***ClosedResourceError*** – if the send stream has been explicitly closed
- ***BrokenResourceError*** – if this stream has been rendered unusable due to external causes

Return type `None`

async send_eof()

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type `None`

class `anyio.streams.tls.TLSAttribute`

Bases: `anyio.TypedAttributeSet`

Contains Transport Layer Security related attributes.

alpn_protocol: `Optional[str]` = <object object>
the selected ALPN protocol

channel_binding_tls_unique: `bytes` = <object object>
the channel binding for type `tls-unique`

cipher: `Tuple[str, str, int]` = <object object>
the selected cipher

peer_certificate: `Optional[Dict[str, Union[str, tuple]]]` = <object object>
the peer certificate in dictionary form (see `ssl.SSLSocket.getpeercert()` for more information)

peer_certificate_binary: `Optional[bytes]` = <object object>
the peer certificate in binary form

server_side: `bool` = <object object>
True if this is the server side of the connection

shared_ciphers: `List[Tuple[str, str, int]]` = <object object>
ciphers shared between both ends of the TLS connection

ssl_object: `ssl.SSLObject` = <object object>
the `SSLObject` used for encryption

standard_compatible: `bool` = `<object object>`

True if this stream does (and expects) a closing TLS handshake when the stream is being closed

tls_version: `str` = `<object object>`

the TLS protocol version (e.g. `TLSv1.2`)

class `anyio.streams.tls.TLSStream`(*transport_stream*, *standard_compatible*, *_ssl_object*,
_read_bio, *_write_bio*)

Bases: `anyio.abc.ByteStream`

A stream wrapper that encrypts all sent data and decrypts received data.

This class has no public initializer; use `wrap()` instead. All extra attributes from `TLSAttribute` are supported.

Variables `transport_stream` (`AnyByteStream`) – the wrapped stream

async `aclose()`

Close the resource.

Return type `None`

property `extra_attributes`

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

async `receive` (*max_bytes=65536*)

Receive at most `max_bytes` bytes from the peer.

Parameters `max_bytes` (`int`) – maximum number of bytes to receive

Return type `bytes`

Returns the received bytes

Raises `EndOfStream` – if this stream has been closed from the other end

async `send` (*item*)

Send the given bytes to the peer.

Parameters `item` (`bytes`) – the bytes to send

Return type `None`

async `send_eof()`

Send an end-of-file indication to the peer.

You should not try to send any further data to this stream after calling this method. This method is idempotent (does nothing on successive calls).

Return type `None`

async `unwrap()`

Does the TLS closing handshake.

Return type `Tuple[Union[ObjectStream[bytes], ByteStream], bytes]`

Returns a tuple of (wrapped byte stream, bytes left in the read buffer)

async classmethod `wrap` (*transport_stream*, ***, *server_side=None*, *hostname=None*,
ssl_context=None, *standard_compatible=True*)

Wrap an existing stream with Transport Layer Security.

This performs a TLS handshake with the peer.

Parameters

- **transport_stream** (`Union[ObjectStream[bytes], ByteStream]`) – a bytes-transporting stream to wrap
- **server_side** (`Optional[bool]`) – True if this is the server side of the connection, False if this is the client side (if omitted, will be set to False if `hostname` has been provided, False otherwise). Used only to create a default context when an explicit context has not been provided.
- **hostname** (`Optional[str]`) – host name of the peer (if host name checking is desired)
- **ssl_context** (`Optional[SSLContext]`) – the `SSLContext` object to use (if not provided, a secure default will be created)
- **standard_compatible** (`bool`) – if False, skip the closing handshake when closing the connection, and don't raise an exception if the peer does the same

Raises **SSLError** – if the TLS handshake fails

Return type `TLSSStream`

```
class anyio.streams.tls.TLSListener(listener, ssl_context, standard_compatible=True, handshake_timeout=30)
```

Bases: `anyio.abc.Listener[anyio.streams.tls.TLSSStream]`

A convenience listener that wraps another listener and auto-negotiates a TLS session on every accepted connection.

If the TLS handshake times out or raises an exception, `handle_handshake_error()` is called to do whatever post-mortem processing is deemed necessary.

Supports only the `standard_compatible` extra attribute.

Parameters

- **listener** (`Listener`) – the listener to wrap
- **ssl_context** (`SSLContext`) – the SSL context object
- **standard_compatible** (`bool`) – a flag passed through to `TLSSStream.wrap()`
- **handshake_timeout** (`float`) – time limit for the TLS handshake (passed to `fail_after()`)

```
async aclose()
```

Close the resource.

Return type `None`

```
property extra_attributes
```

A mapping of the extra attributes to callables that return the corresponding values.

If the provider wraps another provider, the attributes from that wrapper should also be included in the returned mapping (but the wrapper may override the callables from the wrapped instance).

```
async serve(handler, task_group=None)
```

Accept incoming connections as they come in and spawn tasks to handle them.

Parameters

- **handler** (`Callable[[TLSSStream], Any]`) – a callable that will be used to handle each accepted connection

- **task_group** (`Optional[TaskGroup]`) – the task group that will be used to spawn tasks for handling each accepted connection (if omitted, an ad-hoc task group will be created)

Return type `None`

1.13.9 Sockets and networking

async `anyio.connect_tcp`(*remote_host*, *remote_port*, *, *local_host*=None, *tls*=False, *ssl_context*=None, *tls_standard_compatible*=True, *tls_hostname*=None, *happy_eyeballs_delay*=0.25)

Connect to a host using the TCP protocol.

This function implements the stateless version of the Happy Eyeballs algorithm (RFC 6555). If *address* is a host name that resolves to multiple IP addresses, each one is tried until one connection attempt succeeds. If the first attempt does not connect within 250 milliseconds, a second attempt is started using the next address in the list, and so on. On IPv6 enabled systems, an IPv6 address (if available) is tried first.

When the connection has been established, a TLS handshake will be done if either *ssl_context* or *tls_hostname* is not None, or if *tls* is True.

Parameters

- **remote_host** – the IP address or host name to connect to
- **remote_port** – port on the target host to connect to
- **local_host** – the interface address or name to bind the socket to before connecting
- **tls** – True to do a TLS handshake with the connected stream and return a `TLSSStream` instead
- **ssl_context** – the SSL context object to use (if omitted, a default context is created)
- **tls_standard_compatible** – If True, performs the TLS shutdown handshake before closing the stream and requires that the server does this as well. Otherwise, `SSLEOFError` may be raised during reads from the stream. Some protocols, such as HTTP, require this option to be False. See `wrap_socket()` for details.
- **tls_hostname** – host name to check the server certificate against (defaults to the value of *remote_host*)
- **happy_eyeballs_delay** – delay (in seconds) before starting the next connection attempt

Returns a socket stream object if no TLS handshake was done, otherwise a TLS stream

Raises `OSError` – if the connection attempt fails

async `anyio.connect_unix`(*path*)

Connect to the given UNIX socket.

Not available on Windows.

Parameters *path* (`Union[str, PathLike]`) – path to the socket

Return type `SocketStream`

Returns a socket stream object

```
async anyio.create_tcp_listener(*, local_host=None, local_port=0, family=<AddressFamily.AF_UNSPEC: 0>, backlog=65536, reuse_port=False)
```

Create a TCP socket listener.

Parameters

- **local_port** (`int`) – port number to listen on
- **local_host** (`Union[str, IPv4Address, IPv6Address, None]`) – IP address of the interface to listen on. If omitted, listen on all IPv4 and IPv6 interfaces. To listen on all interfaces on a specific address family, use `0.0.0.0` for IPv4 or `::` for IPv6.
- **family** (`Literal[<AddressFamily.AF_UNSPEC: 0>, <AddressFamily.AF_INET: 2>, <AddressFamily.AF_INET6: 10>]`) – address family (used if `interface` was omitted)
- **backlog** (`int`) – maximum number of queued incoming connections (up to a maximum of 2^{16} , or 65536)
- **reuse_port** (`bool`) – True to allow multiple sockets to bind to the same address/port (not supported on Windows)

Return type `MultiListener[SocketStream[Tuple[str, int]]]`

Returns a list of listener objects

```
async anyio.create_unix_listener(path, *, mode=None, backlog=65536)
```

Create a UNIX socket listener.

Not available on Windows.

Parameters

- **path** (`Union[str, PathLike]`) – path of the socket
- **mode** (`Optional[int]`) – permissions to set on the socket
- **backlog** (`int`) – maximum number of queued incoming connections (up to a maximum of 2^{16} , or 65536)

Return type `SocketListener[str]`

Returns a listener object

```
async anyio.create_udp_socket(family=<AddressFamily.AF_UNSPEC: 0>, *, local_host=None, local_port=0, reuse_port=False)
```

Create a UDP socket.

If `port` has been given, the socket will be bound to this port on the local machine, making this socket suitable for providing UDP based services.

Parameters

- **family** (`Literal[<AddressFamily.AF_UNSPEC: 0>, <AddressFamily.AF_INET: 2>, <AddressFamily.AF_INET6: 10>]`) – address family (AF_INET or AF_INET6) – automatically determined from `local_host` if omitted
- **local_host** (`Union[str, IPv4Address, IPv6Address, None]`) – IP address or host name of the local interface to bind to
- **local_port** (`int`) – local port to bind to
- **reuse_port** (`bool`) – True to allow multiple sockets to bind to the same address/port (not supported on Windows)

Return type `UDPSocket`

Returns a UDP socket

async `anyio.create_connected_udp_socket` (*remote_host*, *remote_port*, *, *family*=`<AddressFamily.AF_UNSPEC: 0>`, *local_host*=`None`, *local_port*=`0`, *reuse_port*=`False`)

Create a connected UDP socket.

Connected UDP sockets can only communicate with the specified remote host/port, and any packets sent from other sources are dropped.

Parameters

- **remote_host** (`Union[str, IPv4Address, IPv6Address]`) – remote host to set as the default target
- **remote_port** (`int`) – port on the remote host to set as the default target
- **family** (`Literal[<AddressFamily.AF_UNSPEC: 0>, <AddressFamily.AF_INET: 2>, <AddressFamily.AF_INET6: 10>]`) – address family (`AF_INET` or `AF_INET6`) – automatically determined from *local_host* or *remote_host* if omitted
- **local_host** (`Union[str, IPv4Address, IPv6Address, None]`) – IP address or host name of the local interface to bind to
- **local_port** (`int`) – local port to bind to
- **reuse_port** (`bool`) – True to allow multiple sockets to bind to the same address/port (not supported on Windows)

Return type `ConnectedUDPSocket`

Returns a connected UDP socket

async `anyio.getaddrinfo` (*host*, *port*, *, *family*=`0`, *type*=`0`, *proto*=`0`, *flags*=`0`)

Look up a numeric IP address given a host name.

Internationalized domain names are translated according to the (non-transitional) IDNA 2008 standard.

Note: 4-tuple IPv6 socket addresses are automatically converted to 2-tuples of (host, port), unlike what `socket.getaddrinfo()` does.

Parameters

- **host** (`Union[bytearray, bytes, str]`) – host name
- **port** (`Union[str, int, None]`) – port number
- **family** (`int`) – socket family (`'AF_INET', ...`)
- **type** (`int`) – socket type (`SOCK_STREAM, ...`)
- **proto** (`int`) – protocol number
- **flags** (`int`) – flags to pass to upstream `getaddrinfo()`

Return type `List[Tuple[AddressFamily, SocketKind, int, str, Tuple[str, int]]]`

Returns list of tuples containing (family, type, proto, canonname, sockaddr)

See also:

`socket.getaddrinfo()`

`anyio.getnameinfo(sockaddr, flags=0)`

Look up the host name of an IP address.

Parameters

- **sockaddr** (`Tuple[str, int]`) – socket address (e.g. (ipaddress, port) for IPv4)
- **flags** (`int`) – flags to pass to upstream `getnameinfo()`

Return type `Awaitable[Tuple[str, str]]`

Returns a tuple of (host name, service name)

See also:

`socket.getnameinfo()`

`anyio.wait_socket_readable(sock)`

Wait until the given socket has data to be read.

This does **NOT** work on Windows when using the asyncio backend with a proactor event loop (default on py3.8+).

Warning: Only use this on raw sockets that have not been wrapped by any higher level constructs like socket streams!

Parameters **sock** (`socket`) – a socket object

Raises

- `ClosedResourceError` – if the socket was closed while waiting for the socket to become readable
- `BusyResourceError` – if another task is already waiting for the socket to become readable

Return type `Awaitable[None]`

`anyio.wait_socket_writable(sock)`

Wait until the given socket can be written to.

This does **NOT** work on Windows when using the asyncio backend with a proactor event loop (default on py3.8+).

Warning: Only use this on raw sockets that have not been wrapped by any higher level constructs like socket streams!

Parameters **sock** (`socket`) – a socket object

Raises

- `ClosedResourceError` – if the socket was closed while waiting for the socket to become writable
- `BusyResourceError` – if another task is already waiting for the socket to become writable

Return type `Awaitable[None]`

```

class anyio.abc.SocketAttribute
    Bases: anyio.TypedAttributeSet

class anyio.abc.SocketStream
    Bases: Generic[typing.T_SockAddr], anyio.abc.ByteStream, anyio.abc.sockets.
        _SocketProvider[typing.T_SockAddr]

    Transports bytes over a socket.

    Supports all relevant extra attributes from SocketAttribute.

class anyio.abc.SocketListener
    Bases: Generic[typing.T_SockAddr], anyio.abc.streams.Listener[anyio.
        abc.sockets.SocketStream[typing.T_SockAddr]], anyio.abc.sockets.
        _SocketProvider[typing.T_SockAddr]

    Listens to incoming socket connections.

    Supports all relevant extra attributes from SocketAttribute.

abstract async accept ()
    Accept an incoming connection.

    Return type SocketStream[~T_SockAddr]

async serve (handler, task_group=None)
    Accept incoming connections as they come in and spawn tasks to handle them.

    Parameters

    • handler (Callable[[~T_Stream], Any]) – a callable that will be used to handle each
        accepted connection

    • task_group (Optional[TaskGroup]) – the task group that will be used to spawn
        tasks for handling each accepted connection (if omitted, an ad-hoc task group will be
        created)

    Return type None

class anyio.abc.UDPSocket
    Bases: anyio.abc.streams.UnreliableObjectStream[Tuple[bytes, Tuple[str, int]]],
        anyio.abc.sockets._SocketProvider[Tuple[str, int]]

    Represents an unconnected UDP socket.

    Supports all relevant extra attributes from SocketAttribute.

async sendto (data, host, port)
    Alias for send() ((data, (host, port))).

    Return type None

class anyio.abc.ConnectedUDPSocket
    Bases: anyio.abc.streams.UnreliableObjectStream[bytes], anyio.abc.sockets.
        _SocketProvider[Tuple[str, int]]

    Represents an connected UDP socket.

    Supports all relevant extra attributes from SocketAttribute.

```

1.13.10 Subprocesses

async `anyio.run_process(command, *, input=None, stdout=-1, stderr=-1, check=True)`

Run an external command in a subprocess and wait until it completes.

See also:

`subprocess.run()`

Parameters

- **command** (`Union[str, Sequence[str]]`) – either a string to pass to the shell, or an iterable of strings containing the executable name or path and its arguments
- **input** (`Optional[bytes]`) – bytes passed to the standard input of the subprocess
- **stdout** (`int`) – either `subprocess.PIPE` or `subprocess.DEVNULL`
- **stderr** (`int`) – one of `subprocess.PIPE`, `subprocess.DEVNULL` or `subprocess.STDOUT`
- **check** (`bool`) – if `True`, raise `CalledProcessError` if the process terminates with a return code other than 0

Return type `CompletedProcess`

Returns an object representing the completed process

Raises `CalledProcessError` – if `check` is `True` and the process exits with a nonzero return code

async `anyio.open_process(command, *, stdin=-1, stdout=-1, stderr=-1)`

Start an external command in a subprocess.

See also:

`subprocess.Popen`

Parameters

- **command** (`Union[str, Sequence[str]]`) – either a string to pass to the shell, or an iterable of strings containing the executable name or path and its arguments
- **stdin** (`int`) – either `subprocess.PIPE` or `subprocess.DEVNULL`
- **stdout** (`int`) – either `subprocess.PIPE` or `subprocess.DEVNULL`
- **stderr** (`int`) – one of `subprocess.PIPE`, `subprocess.DEVNULL` or `subprocess.STDOUT`

Return type `Process`

Returns an asynchronous process object

class `anyio.abc.Process`

Bases: `anyio.abc.AsyncResource`

An asynchronous version of `subprocess.Popen`.

abstract `kill()`

Kills the process.

On Windows, this calls `TerminateProcess()`. On POSIX systems, this sends `SIGKILL` to the process.

See also:

```
subprocess.Popen.kill()
```

Return type `None`

abstract property pid

The process ID of the process.

Return type `int`

abstract property returncode

The return code of the process. If the process has not yet terminated, this will be `None`.

Return type `Optional[int]`

abstract send_signal(*signal*)

Send a signal to the subprocess.

See also:

```
subprocess.Popen.send_signal()
```

Parameters **signal** (`int`) – the signal number (e.g. `signal.SIGHUP`)

Return type `None`

abstract property stderr

The stream for the standard error output of the process.

Return type `Optional[ByteReceiveStream]`

abstract property stdin

The stream for the standard input of the process.

Return type `Optional[ByteSendStream]`

abstract property stdout

The stream for the standard output of the process.

Return type `Optional[ByteReceiveStream]`

abstract terminate()

Terminates the process, gracefully if possible.

On Windows, this calls `TerminateProcess()`. On POSIX systems, this sends `SIGTERM` to the process.

See also:

```
subprocess.Popen.terminate()
```

Return type `None`

abstract async wait()

Wait until the process exits.

Return type `int`

Returns the exit code of the process

1.13.11 Synchronization

`anyio.create_semaphore(value)`

Create an asynchronous semaphore.

Parameters `value` (`int`) – the semaphore’s initial value

Return type `Semaphore`

Returns a semaphore object

`anyio.create_lock()`

Create an asynchronous lock.

Return type `Lock`

Returns a lock object

`anyio.create_event()`

Create an asynchronous event object.

Return type `Event`

Returns an event object

`anyio.create_condition(lock=None)`

Create an asynchronous condition.

Parameters `lock` (`Optional[Lock]`) – the lock to base the condition object on

Return type `Condition`

Returns a condition object

`anyio.create_capacity_limiter(total_tokens)`

Create a capacity limiter.

Parameters `total_tokens` (`float`) – the total number of tokens available for borrowing (can be an integer or `math.inf`)

Return type `CapacityLimiter`

Returns a capacity limiter object

class `anyio.abc.Semaphore`

Bases: `object`

abstract async acquire()

Decrement the semaphore value, blocking if necessary.

Return type `None`

abstract async release()

Increment the semaphore value.

Return type `None`

abstract property value

The current value of the semaphore.

Return type `int`

class `anyio.abc.Lock`

Bases: `object`

abstract async acquire()

Acquire the lock.

Return type `None`

abstract locked()

Return True if the lock is currently held.

Return type `bool`

abstract async release()

Release the lock.

Return type `None`

class `anyio.abc.Event`

Bases: `object`

abstract is_set()

Return True if the flag is set, False if not.

Return type `bool`

abstract async set()

Set the flag, notifying all listeners.

Return type `None`

abstract async wait()

Wait until the flag has been set.

If the flag has already been set when this method is called, it returns immediately.

Return type `bool`

class `anyio.abc.Condition`

Bases: `object`

abstract async acquire()

Acquire the underlying lock.

Return type `None`

abstract locked()

Return True if the lock is set.

Return type `bool`

abstract async notify(*n=1*)

Notify exactly *n* listeners.

Return type `None`

abstract async notify_all()

Notify all the listeners.

Return type `None`

abstract async release()

Release the underlying lock.

Return type `None`

abstract async wait()

Wait for a notification.

Return type `None`

class `anyio.abc.CapacityLimiter`

Bases: `object`

abstract async acquire()

Acquire a token for the current task, waiting if necessary for one to become available.

Return type `None`

abstract async acquire_nowait()

Acquire a token for the current task without waiting for one to become available.

Raises `WouldBlock` – if there are no tokens available for borrowing

Return type `None`

abstract async acquire_on_behalf_of(borrower)

Acquire a token, waiting if necessary for one to become available.

Parameters `borrower` – the entity borrowing a token

Return type `None`

abstract async acquire_on_behalf_of_nowait(borrower)

Acquire a token without waiting for one to become available.

Parameters `borrower` – the entity borrowing a token

Raises `WouldBlock` – if there are no tokens available for borrowing

Return type `None`

abstract property available_tokens

The number of tokens currently available to be borrowed

Return type `float`

abstract property borrowed_tokens

The number of tokens that have currently been borrowed.

Return type `int`

abstract async release()

Release the token held by the current task. :raises `RuntimeError`: if the current task has not borrowed a token from this limiter.

Return type `None`

abstract async release_on_behalf_of(borrower)

Release the token held by the given borrower.

Raises `RuntimeError` – if the borrower has not borrowed a token from this limiter.

Return type `None`

abstract async set_total_tokens(value)

Set the total number of tokens.

If the total number of tokens is increased, the proportionate number of tasks waiting on this limiter will be granted their tokens.

Parameters `value` (`float`) – the new total number of tokens (≥ 1)

Return type `None`

abstract property total_tokens

The total number of tokens available for borrowing.

Return type `float`

1.13.12 Operating system signals

`anyio.open_signal_receiver(*signals)`

Start receiving operating system signals.

Parameters `signals` (`int`) – signals to receive (e.g. `signal.SIGINT`)

Return type `AsyncContextManager[AsyncIterator[int]]`

Returns an asynchronous context manager for an asynchronous iterator which yields signal numbers

Warning: Windows does not support signals natively so it is best to avoid relying on this in cross-platform applications.

1.13.13 Compatibility

async `anyio.maybe_async(__obj)`

Await on the given object if necessary.

This function is intended to bridge the gap between AnyIO 2.x and 3.x where some functions and methods were converted from coroutine functions into regular functions.

Returns the result of awaiting on the object if coroutine, or the object itself otherwise

New in version 2.2.

`anyio.maybe_async_cm(cm)`

Wrap a regular context manager as an async one if necessary.

This function is intended to bridge the gap between AnyIO 2.x and 3.x where some functions and methods were changed to return regular context managers instead of async ones.

Parameters `cm` (`Union[ContextManager[~T], AsyncContextManager[~T]]`) – a regular or async context manager

Return type `AsyncContextManager[~T]`

Returns an async context manager

New in version 2.2.

1.13.14 Testing and debugging

class `anyio.TaskInfo(id, parent_id, name, coro)`

Bases: `object`

Represents an asynchronous task.

Variables

- `id` (`int`) – the unique identifier of the task
- `parent_id` (`Optional[int]`) – the identifier of the parent task, if any
- `name` (`str`) – the description of the task (if any)
- `coro` (`Coroutine`) – the coroutine object of the task

async `anyio.get_current_task()`

Return the current task.

Return type `TaskInfo`

Returns a representation of the current task

async `anyio.get_running_tasks()`

Return a list of running tasks in the current event loop.

Return type `List[TaskInfo]`

Returns a list of task info objects

async `anyio.wait_all_tasks_blocked()`

Wait until all other tasks are waiting for something.

Return type `None`

1.13.15 Exceptions

exception `anyio.BrokenResourceError`

Bases: `Exception`

Raised when trying to use a resource that has been rendered unusable due to external causes (e.g. a send stream whose peer has disconnected).

exception `anyio.BusyResourceError` (*action*)

Bases: `Exception`

Raised when two tasks are trying to read from or write to the same resource concurrently.

exception `anyio.ClosedResourceError`

Bases: `Exception`

Raised when trying to use a resource that has been closed.

exception `anyio.DelimiterNotFound` (*max_bytes*)

Bases: `Exception`

Raised during `receive_until()` if the maximum number of bytes has been read without the delimiter being found.

exception `anyio.EndOfStream`

Bases: `Exception`

Raised when trying to read from a stream that has been closed from the other end.

exception `anyio.ExceptionGroup`

Bases: `BaseException`

Raised when multiple exceptions have been raised in a task group.

exception `anyio.IncompleteRead`

Bases: `Exception`

Raised during `receive_exactly()` or `receive_until()` if the connection is closed before the requested amount of bytes has been read.

exception `anyio.TypedAttributeLookupError`

Bases: `LookupError`

Raised by `extra()` when the given typed attribute is not found and no default value has been given.

exception `anyio.WouldBlock`

Bases: `Exception`

Raised by `X_nowait` functions if `X()` would block.

1.14 Getting help

If you are having trouble with AnyIO, you have a couple ways for getting support:

- Post a question on [Stack Overflow](#) and use the `anyio` tag
- Join the [python-trio/AnyIO](#) room on Gitter

1.15 Reporting bugs

If you're fairly certain that you have discovered a bug, you can [file an issue](#) on Github. If you feel unsure, come talk to us first! The issue tracker is **not** the proper venue for asking support questions.

1.16 Contributing to AnyIO

If you wish to contribute a fix or feature to AnyIO, please follow the following guidelines.

When you make a pull request against the main AnyIO codebase, Github runs the AnyIO test suite against your modified code. Before making a pull request, you should ensure that the modified code passes tests locally. To that end, the use of `tox` is recommended. The default `tox` run first runs code style fixing tools and then the actual test suite. To only run the code style fixers, run `tox -e lint`. To run the checks on all environments in parallel, invoke `tox` with `tox -p`.

The use of `pre-commit` is also highly recommended. You can find a sample configuration file at the root of the repository.

1.16.1 Making a pull request on Github

To get your changes merged to the main codebase, you need a Github account.

1. Fork the repository (if you don't have your own fork of it yet) by navigating to the [main AnyIO repository](#) and clicking on "Fork" near the top right corner.
2. Clone the forked repository to your local machine with `git clone git@github.com:yourusername/anyio`.
3. Create a branch for your pull request, like `git checkout -b myfixname`
4. Make the desired changes to the code base.
5. Commit your changes locally. If your changes close an existing issue, add the text `Fixes XXX.` or `Closes XXX.` to the commit message (where XXX is the issue number).
6. Push the changeset(s) to your forked repository (`git push`)
7. Navigate to Pull requests page on the original repository (not your fork) and click "New pull request"
8. Click on the text "compare across forks".
9. Select your own fork as the head repository and then select the correct branch name.
10. Click on "Create pull request".

If you have trouble, consult the [pull request making guide](#) on [opensource.com](#).

1.17 Version history

This library adheres to [Semantic Versioning 2.0](#).

2.2.0

- Added the `maybe_async()` and `maybe_async_cm()` functions to facilitate forward compatibility with AnyIO 3
- Fixed socket stream bug on asyncio where receiving a half-close from the peer would shut down the entire connection
- Fixed native task names not being set on asyncio on Python 3.8+
- Fixed `TLSSStream.send_eof()` raising `ValueError` instead of the expected `NotImplementedError`
- Fixed `open_signal_receiver()` on asyncio and curio hanging if the cancel scope was cancelled before the function could run
- Fixed Trio test runner causing unwarranted test errors on `BaseException`s` (PR by Matthias Urlichs)
- Fixed formatted output of `ExceptionGroup` containing too many newlines

2.1.0

- Added the `spawn_task()` and `wrap_async_context_manager()` methods to `BlockingPortal`
- Added the `handshake_timeout` and `error_handler` parameters to `TLSListener`
- Fixed `Event` objects on the trio backend not inheriting from `anyio.abc.Event`
- Fixed `run_sync_in_worker_thread()` raising `UnboundLocalError` on asyncio when cancelled
- Fixed `send()` on socket streams not raising any exception on asyncio, and an unwrapped `BrokenPipeError` on trio and curio when the peer has disconnected
- Fixed `MemoryObjectSendStream.send()` raising `BrokenResourceError` when the last receiver is closed right after receiving the item
- Fixed `ValueError: Invalid file descriptor: -1` when closing a `SocketListener` on asyncio

2.0.2

- Fixed one more case of `AttributeError: 'async_generator_asend' object has no attribute 'cr_await'` on asyncio

2.0.1

- Fixed broken `MultiListener.extra()` (PR by daa)
- Fixed `TLSSStream` returning an empty bytes object instead of raising `EndOfStream` when trying to receive from the stream after a closing handshake
- Fixed `AttributeError` when cancelling a task group's scope inside an async test fixture on asyncio
- Fixed `wait_all_tasks_blocked()` raising `AttributeError` on asyncio if a native task is waiting on an async generator's `asend()` method

2.0.0

- General new features:
 - Added support for subprocesses
 - Added support for “blocking portals” which allow running functions in the event loop thread from external threads

- Added the `anyio.aclose_forcefully()` function for closing asynchronous resources as quickly as possible
- General changes/fixes:
 - **BACKWARDS INCOMPATIBLE** Some functions have been renamed or removed (see further below for socket/fileio API changes):
 - * `finalize()` → (removed; use `async_generator.aclosing()` instead)
 - * `receive_signals()` → `open_signal_receiver()`
 - * `run_in_thread()` → `run_sync_in_worker_thread()`
 - * `current_default_thread_limiter()` → `current_default_worker_thread_limiter()`
 - * `ResourceBusyError` → `BusyResourceError`
 - **BACKWARDS INCOMPATIBLE** Exception classes were moved to the top level package
 - Dropped support for Python 3.5
 - Bumped minimum versions of trio and curio to v0.16 and v1.4, respectively
 - Changed the `repr()` of `ExceptionGroup` to match trio's `MultiError`
- Backend specific changes and fixes:
 - `asyncio`: Added support for `ProactorEventLoop`. This allows `asyncio` applications to use AnyIO on Windows even without using AnyIO as the entry point.
 - `asyncio`: The `asyncio` backend now uses `asyncio.run()` behind the scenes which properly shuts down `async` generators and cancels any leftover native tasks
 - `curio`: Worked around the limitation where a task can only be cancelled twice (any cancellations beyond that were ignored)
 - `asyncio` + `curio`: a cancellation check now calls `sleep(0)`, allowing the scheduler to switch to a different task
 - `asyncio` + `curio`: Host name resolution now uses [IDNA 2008](#) (with UTS 46 compatibility mapping, just like trio)
 - `asyncio` + `curio`: Fixed a bug where a task group would abandon its subtasks if its own cancel scope was cancelled while it was waiting for subtasks to finish
 - `asyncio` + `curio`: Fixed recursive tracebacks when a single exception from an inner task group is reraised in an outer task group
- Socket/stream changes:
 - **BACKWARDS INCOMPATIBLE** The stream class structure was completely overhauled. There are now separate abstract base classes for receive and send streams, byte streams and reliable and unreliable object streams. Stream wrappers are much better supported by this new ABC structure and a new “typed extra attribute” system that lets you query the wrapper chain for the attributes you want via `.extra(...)`.
 - **BACKWARDS INCOMPATIBLE** Socket server functionality has been refactored into a network-agnostic listener system
 - **BACKWARDS INCOMPATIBLE** TLS functionality has been split off from `SocketStream` and can now work over any bidirectional bytes-based stream – you can now establish a TLS encrypted communications pathway over UNIX sockets or even memory object streams. The `TLSRequired` exception has also been removed as it is no longer necessary.

- **BACKWARDS INCOMPATIBLE** Buffering functionality (`receive_until()` and `receive_exactly()`) was split off from `SocketStream` into a stream wrapper class (`anyio.streams.buffered.BufferedByteReceiveStream`)
- **BACKWARDS INCOMPATIBLE** IPv6 addresses are now reported as 2-tuples. If original 4-tuple form contains a nonzero scope ID, it is appended to the address with `%` as the separator.
- **BACKWARDS INCOMPATIBLE** Byte streams (including socket streams) now raise `EndOfStream` instead of returning an empty bytes object when the stream has been closed from the other end
- **BACKWARDS INCOMPATIBLE** The socket API has changes:
 - * `create_tcp_server()` → `create_tcp_listener()`
 - * `create_unix_server()` → `create_unix_listener()`
 - * `create_udp_socket()` had some of its parameters changed:
 - `interface` → `local_address`
 - `port` → `local_port`
 - `reuse_address` was replaced with `reuse_port` (and sets `SO_REUSEPORT` instead of `SO_REUSEADDR`)
 - * `connect_tcp()` had some of its parameters changed:
 - `address` → `remote_address`
 - `port` → `remote_port`
 - `bind_host` → `local_address`
 - `bind_port` → (removed)
 - `autostart_tls` → `tls`
 - `tls_hostname` (new parameter, when you want to match the certificate against something else than `remote_address`)
 - * `connect_tcp()` now returns a `TLSStream` if TLS was enabled
 - * `notify_socket_closing()` was removed, as it is no longer used by AnyIO
 - * `SocketStream` has changes to its methods and attributes:
 - `address` → `.extra(SocketAttribute.local_address)`
 - `alpn_protocol` → `.extra(TLSAttribute.alpn_protocol)`
 - `close()` → `aclose()`
 - `get_channel_binding` → `.extra(TLSAttribute.channel_binding_tls_unique)`
 - `cipher` → `.extra(TLSAttribute.cipher)`
 - `getpeercert` → `.extra(SocketAttribute.peer_certificate)` or `.extra(SocketAttribute.peer_certificate_binary)`
 - `getsockopt()` → `.extra(SocketAttribute.raw_socket).getsockopt(...)`
 - `peer_address` → `.extra(SocketAttribute.remote_address)`
 - `receive_chunks()` → (removed; use `async for` on the stream instead)
 - `receive_delimited_chunks()` → (removed)
 - `receive_exactly()` → `BufferedReceiveStream.receive_exactly()`

- `receive_some()` → `receive()`
- `receive_until()` → `BufferedReceiveStream.receive_until()`
- `send_all()` → `send()`
- `setsockopt()` → `.extra(SocketAttribute.raw_socket).setsockopt(...)`
- `shared_ciphers` → `.extra(TLSAttribute.shared_ciphers)`
- `server_side` → `.extra(TLSAttribute.server_side)`
- `start_tls()` → `stream = TLSStream.wrap(...)`
- `tls_version` → `.extra(TLSAttribute.tls_version)`
- * **UDPSocket** has changes to its methods and attributes:
 - `address` → `.extra(SocketAttribute.local_address)`
 - `getsockopt()` → `.extra(SocketAttribute.raw_socket).getsockopt(...)`
 - `port` → `.extra(SocketAttribute.local_port)`
 - `receive()` no longer takes a maximum bytes argument
 - `receive_packets()` → (removed; use `async for` on the UDP socket instead)
 - `send()` → requires a tuple for destination now (address, port), for compatibility with the new `UnreliableObjectStream` interface. The `sendto()` method works like the old `send()` method.
 - `setsockopt()` → `.extra(SocketAttribute.raw_socket).setsockopt(...)`
- **BACKWARDS INCOMPATIBLE** Renamed the `max_size` parameter to `max_bytes` wherever it occurred (this was inconsistently named `max_bytes` in some subclasses before)
- Added memory object streams as a replacement for queues
- Added stream wrappers for encoding/decoding unicode strings
- Support for the `SO_REUSEPORT` option (allows binding more than one socket to the same address/port combination, as long as they all have this option set) has been added to TCP listeners and UDP sockets
- The `send_eof()` method was added to all (bidirectional) streams
- File I/O changes:
 - **BACKWARDS INCOMPATIBLE** Asynchronous file I/O functionality now uses a common code base (`anyio.AsyncFile`) instead of backend-native classes
 - **BACKWARDS INCOMPATIBLE** The File I/O API has changes to its functions and methods:
 - * `aopen()` → `open_file()`
 - * `AsyncFileclose()` → `AsyncFileaclose()`
- Task synchronization changes:
 - **BACKWARDS INCOMPATIBLE** Queues were replaced by memory object streams
 - **BACKWARDS INCOMPATIBLE** Added the `acquire()` and `release()` methods to the `Lock`, `Condition` and `Semaphore` classes
 - **BACKWARDS INCOMPATIBLE** Removed the `Event.clear()` method. You must now replace the event object with a new one rather than clear the old one.
 - Fixed `Condition.wait()` not working on `asyncio` and `curio` (PR by Matt Westcott)

- Testing changes:
 - **BACKWARDS INCOMPATIBLE** Removed the `--anyio-backends` command line option for the pytest plugin. Use the `-k` option to do ad-hoc filtering, and the `anyio_backend` fixture to control which backends you wish to run the tests by default.
 - The pytest plugin was refactored to run the test and all its related async fixtures inside the same event loop, making async fixtures much more useful
 - Fixed Hypothesis support in the pytest plugin (it was not actually running the Hypothesis tests at all)

1.4.0

- Added async name resolution functions (`anyio.getaddrinfo()` and `anyio.getnameinfo()`)
- Added the `family` and `reuse_address` parameters to `anyio.create_udp_socket()` (Enables multicast support; test contributed by Matthias Urlichs)
- Fixed `fail.after(0)` not raising a timeout error on `asyncio` and `curio`
- Fixed `move_on_after()` and `fail_after()` getting stuck on `curio` in some circumstances
- Fixed socket operations not allowing timeouts to cancel the task
- Fixed API documentation on `Stream.receive_until()` which claimed that the delimiter will be included in the returned data when it really isn't
- Harmonized the default task names across all backends
- `wait_all_tasks_blocked()` no longer considers tasks waiting on `sleep(0)` to be blocked on `asyncio` and `curio`
- Fixed the type of the `address` parameter in `UDPSocket.send()` to include `IPAddress` objects (which were already supported by the backing implementation)
- Fixed `UDPSocket.send()` to resolve host names using `anyio.getaddrinfo()` before calling `socket.sendto()` to avoid blocking on synchronous name resolution
- Switched to using `anyio.getaddrinfo()` for name lookups

1.3.1

- Fixed warnings caused by trio 0.15
- Worked around a compatibility issue between `uvloop` and Python 3.9 (missing `shutdown_default_executor()` method)

1.3.0

- Fixed compatibility with Curio 1.0
- Made it possible to assert fine grained control over which AnyIO backends and backend options are being used with each test
- Added the `address` and `peer_address` properties to the `SocketStream` interface

1.2.3

- Repackaged release (v1.2.2 contained extra files from an experimental branch which broke imports)

1.2.2

- Fixed `CancelledError` leaking from a cancel scope on `asyncio` if the task previously received a cancellation exception
- Fixed `AttributeError` when cancelling a generator-based task (`asyncio`)
- Fixed `wait_all_tasks_blocked()` not working with generator-based tasks (`asyncio`)

- Fixed an unnecessary delay in `connect_tcp()` if an earlier attempt succeeds
- Fixed `AssertionError` in `connect_tcp()` if multiple connection attempts succeed simultaneously

1.2.1

- Fixed cancellation errors leaking from a task group when they are contained in an exception group
- Fixed trio v0.13 compatibility on Windows
- Fixed inconsistent queue capacity across backends when capacity was defined as 0 (trio = 0, others = infinite)
- Fixed socket creation failure crashing `connect_tcp()`

1.2.0

- Added the possibility to parametrize regular pytest test functions against the selected list of backends
- Added the `set_total_tokens()` method to `CapacityLimiter`
- Added the `anyio.current_default_thread_limiter()` function
- Added the `cancellable` parameter to `anyio.run_in_thread()`
- Implemented the Happy Eyeballs ([RFC 6555](#)) algorithm for `anyio.connect_tcp()`
- Fixed `KeyError` on `asyncio` and `curio` where entering and exiting a cancel scope happens in different tasks
- Fixed deprecation warnings on Python 3.8 about the `loop` argument of `asyncio.Event()`
- Forced the use `WindowsSelectorEventLoopPolicy` in `asyncio.run` when on Windows and `asyncio` to keep network functionality working
- Worker threads are now spawned with `daemon=True` on all backends, not just trio
- Dropped support for trio v0.11

1.1.0

- Added the `lock` parameter to `anyio.create_condition()` (PR by Matthias Urlichs)
- Added async iteration for queues (PR by Matthias Urlichs)
- Added capacity limiters
- Added the possibility of using capacity limiters for limiting the maximum number of threads
- Fixed compatibility with trio v0.12
- Fixed IPv6 support in `create_tcp_server()`, `connect_tcp()` and `create_udp_socket()`
- Fixed mishandling of task cancellation while the task is running a worker thread on `asyncio` and `curio`

1.0.0

- Fixed `pathlib2` compatibility with `anyio.aopen()`
- Fixed timeouts not propagating from nested scopes on `asyncio` and `curio` (PR by Matthias Urlichs)
- Fixed incorrect call order in socket close notifications on `asyncio` (mostly affecting Windows)
- Prefixed backend module names with an underscore to better indicate privateness

1.0.0rc2

- Fixed some corner cases of cancellation where behavior on `asyncio` and `curio` did not match with that of trio. Thanks to Joshua Oreman for help with this.

- Fixed `current_effective_deadline()` not taking shielded cancellation scopes into account on `asyncio` and `curio`
- Fixed task cancellation not happening right away on `asyncio` and `curio` when a cancel scope is entered when the deadline has already passed
- Fixed exception group containing only cancellation exceptions not being swallowed by a timed out cancel scope on `asyncio` and `curio`
- Added the `current_time()` function
- Replaced `CancelledError` with `get_cancelled_exc_class()`
- Added support for [Hypothesis](#)
- Added support for [PEP 561](#)
- Use `uvloop` for the `asyncio` backend by default when available (but only on CPython)

1.0.0rc1

- Fixed `setsockopt()` passing options to the underlying method in the wrong manner
- Fixed cancellation propagation from nested task groups
- Fixed `get_running_tasks()` returning tasks from other event loops
- Added the `parent_id` attribute to `anyio.TaskInfo`
- Added the `get_current_task()` function
- Added guards to protect against concurrent read/write from/to sockets by multiple tasks
- Added the `notify_socket_close()` function

1.0.0b2

- Added introspection of running tasks via `anyio.get_running_tasks()`
- Added the `getsockopt()` and `setsockopt()` methods to the `SocketStream` API
- Fixed mishandling of large buffers by `BaseSocket.sendall()`
- Fixed compatibility with (and upgraded minimum required version to) `trio v0.11`

1.0.0b1

- Initial release

INDEX

A

`accept()` (*anyio.abc.SocketListener* method), 53
`aclose()` (*anyio.abc.AsyncResource* method), 27
`aclose()` (*anyio.AsyncFile* method), 33
`aclose()` (*anyio.streams.buffered.BufferedByteReceiveStream* method), 39
`aclose()` (*anyio.streams.memory.MemoryObjectReceiveStream* method), 40
`aclose()` (*anyio.streams.memory.MemoryObjectSendStream* method), 41
`aclose()` (*anyio.streams.stapled.MultiListener* method), 42
`aclose()` (*anyio.streams.stapled.StapledByteStream* method), 42
`aclose()` (*anyio.streams.stapled.StapledObjectStream* method), 43
`aclose()` (*anyio.streams.text.TextReceiveStream* method), 44
`aclose()` (*anyio.streams.text.TextSendStream* method), 45
`aclose()` (*anyio.streams.text.TextStream* method), 45
`aclose()` (*anyio.streams.tls.TLSListener* method), 48
`aclose()` (*anyio.streams.tls.TLSStream* method), 47
`aclose_forcefully()` (in module *anyio*), 27
`acquire()` (*anyio.abc.CapacityLimiter* method), 57
`acquire()` (*anyio.abc.Condition* method), 57
`acquire()` (*anyio.abc.Lock* method), 56
`acquire()` (*anyio.abc.Semaphore* method), 56
`acquire_nowait()` (*anyio.abc.CapacityLimiter* method), 58
`acquire_on_behalf_of()` (*anyio.abc.CapacityLimiter* method), 58
`acquire_on_behalf_of_nowait()` (*anyio.abc.CapacityLimiter* method), 58
`alpn_protocol` (*anyio.streams.tls.TLSAttribute* attribute), 46
AnyByteReceiveStream (in module *anyio.abc*), 37
AnyByteSendStream (in module *anyio.abc*), 38
AnyByteStream (in module *anyio.abc*), 39
AnyUnreliableByteReceiveStream (in module *anyio.abc*), 35
AnyUnreliableByteSendStream (in module

anyio.abc), 36

AnyUnreliableByteStream (in module *anyio.abc*), 37

AsyncFile (class in *anyio*), 32

AsyncResource (class in *anyio.abc*), 27

`available_tokens()` (*anyio.abc.CapacityLimiter* property), 58

B

BlockingPortal (class in *anyio.abc*), 31

`borrowed_tokens()` (*anyio.abc.CapacityLimiter* property), 58

BrokenResourceError, 60

`buffer()` (*anyio.streams.buffered.BufferedByteReceiveStream* property), 39

BufferedByteReceiveStream (class in *anyio.streams.buffered*), 39

BusyResourceError, 60

ByteReceiveStream (class in *anyio.abc*), 34

ByteSendStream (class in *anyio.abc*), 34

ByteStream (class in *anyio.abc*), 35

C

`call()` (*anyio.abc.BlockingPortal* method), 31

`cancel()` (*anyio.abc.CancelScope* method), 29

`cancel_called()` (*anyio.abc.CancelScope* property), 29

CancelScope (class in *anyio.abc*), 29

CapacityLimiter (class in *anyio.abc*), 57

`channel_binding_tls_unique` (*anyio.streams.tls.TLSAttribute* attribute), 46

`cipher` (*anyio.streams.tls.TLSAttribute* attribute), 46

`clone()` (*anyio.streams.memory.MemoryObjectReceiveStream* method), 40

`clone()` (*anyio.streams.memory.MemoryObjectSendStream* method), 41

ClosedResourceError, 60

Condition (class in *anyio.abc*), 57

`connect_tcp()` (in module *anyio*), 49

`connect_unix()` (in module *anyio*), 49

ConnectedUDPSocket (class in *anyio.abc*), 53

`create_blocking_portal()` (in module *anyio*), 30

`create_capacity_limiter()` (in module *anyio*), 56

`create_condition()` (in module *anyio*), 56

`create_connected_udp_socket()` (in module *anyio*), 51

`create_event()` (in module *anyio*), 56

`create_lock()` (in module *anyio*), 56

`create_memory_object_stream()` (in module *anyio*), 33

`create_semaphore()` (in module *anyio*), 56

`create_task_group()` (in module *anyio*), 29

`create_tcp_listener()` (in module *anyio*), 49

`create_udp_socket()` (in module *anyio*), 50

`create_unix_listener()` (in module *anyio*), 50

`current_default_worker_thread_limiter()` (in module *anyio*), 30

`current_effective_deadline()` (in module *anyio*), 29

`current_time()` (in module *anyio*), 27

D

`deadline()` (*anyio.abc.CancelScope* property), 29

`DelimiterNotFound`, 60E

`EndOfStream`, 60

`Event` (class in *anyio.abc*), 57

`ExceptionGroup`, 60

`extra()` (*anyio.TypedAttributeProvider* method), 28

`extra_attributes()` (*anyio.streams.buffered.BufferedByteReceiveStream* property), 39

`extra_attributes()` (*anyio.streams.stapled.MultiListener* property), 42

`extra_attributes()` (*anyio.streams.stapled.StapledByteStream* property), 42

`extra_attributes()` (*anyio.streams.stapled.StapledObjectStream* property), 43

`extra_attributes()` (*anyio.streams.text.TextReceiveStream* property), 44

`extra_attributes()` (*anyio.streams.text.TextSendStream* property), 45

`extra_attributes()` (*anyio.streams.text.TextStream* property), 45

`extra_attributes()` (*anyio.streams.tls.TLSListener* property), 48

`extra_attributes()` (*anyio.streams.tls.TLSStream* property), 47

`extra_attributes()` (*anyio.TypedAttributeProvider* property), 28

F

`fail_after()` (in module *anyio*), 28

G

`get_all_backends()` (in module *anyio*), 27

`get_cancelled_exc_class()` (in module *anyio*), 27

`get_current_task()` (in module *anyio*), 59

`get_running_tasks()` (in module *anyio*), 60

`getaddrinfo()` (in module *anyio*), 51

`getnameinfo()` (in module *anyio*), 51

I

`IncompleteRead`, 60

`is_set()` (*anyio.abc.Event* method), 57

K

`kill()` (*anyio.abc.Process* method), 54

L

`Listener` (class in *anyio.abc*), 35

`Lock` (class in *anyio.abc*), 56

`locked()` (*anyio.abc.Condition* method), 57

`locked()` (*anyio.abc.Lock* method), 57

M

`maybe_async()` (in module *anyio*), 59

`maybe_async_cm()` (in module *anyio*), 59

`MemoryObjectReceiveStream` (class in *anyio.streams.memory*), 40

`MemoryObjectSendStream` (class in *anyio.streams.memory*), 41

`move_on_after()` (in module *anyio*), 28

`MultiListener` (class in *anyio.streams.stapled*), 42

N

`notify()` (*anyio.abc.Condition* method), 57

`notify_all()` (*anyio.abc.Condition* method), 57

O

`ObjectReceiveStream` (class in *anyio.abc*), 34

`ObjectSendStream` (class in *anyio.abc*), 34

`ObjectStream` (class in *anyio.abc*), 34

`open_cancel_scope()` (in module *anyio*), 28

`open_file()` (in module *anyio*), 32

`open_process()` (in module *anyio*), 54

`open_signal_receiver()` (in module *anyio*), 59

P

`peer_certificate` (*anyio.streams.tls.TLSAttribute* attribute), 46

`peer_certificate_binary` (*anyio.streams.tls.TLSAttribute* attribute), 46

`pid()` (*anyio.abc.Process* property), 55

Process (class in *anyio.abc*), 54

Python Enhancement Proposals
PEP 561, 68

R

`receive()` (*anyio.abc.ByteReceiveStream* method), 34

`receive()` (*anyio.abc.UnreliableObjectReceiveStream* method), 33

`receive()` (*anyio.streams.buffered.BufferedByteReceiveStream* method), 40

`receive()` (*anyio.streams.memory.MemoryObjectReceiveStream* method), 40

`receive()` (*anyio.streams.stapled.StapledByteStream* method), 42

`receive()` (*anyio.streams.stapled.StapledObjectStream* method), 43

`receive()` (*anyio.streams.text.TextReceiveStream* method), 44

`receive()` (*anyio.streams.text.TextStream* method), 45

`receive()` (*anyio.streams.tls.TLSStream* method), 47

`receive_exactly()` (*anyio.streams.buffered.BufferedByteReceiveStream* method), 40

`receive_nowait()` (*anyio.streams.memory.MemoryObjectReceiveStream* method), 41

`receive_until()` (*anyio.streams.buffered.BufferedByteReceiveStream* method), 40

`release()` (*anyio.abc.CapacityLimiter* method), 58

`release()` (*anyio.abc.Condition* method), 57

`release()` (*anyio.abc.Lock* method), 57

`release()` (*anyio.abc.Semaphore* method), 56

`release_on_behalf_of()` (*anyio.abc.CapacityLimiter* method), 58

`returncode()` (*anyio.abc.Process* property), 55

RFC

RFC 6555, 67

`run()` (in module *anyio*), 26

`run_async_from_thread()` (in module *anyio*), 30

`run_process()` (in module *anyio*), 54

`run_sync_in_worker_thread()` (in module *anyio*), 30

S

Semaphore (class in *anyio.abc*), 56

`send()` (*anyio.abc.ByteSendStream* method), 35

`send()` (*anyio.abc.UnreliableObjectSendStream* method), 33

`send()` (*anyio.streams.memory.MemoryObjectSendStream* method), 41

`send()` (*anyio.streams.stapled.StapledByteStream* method), 43

`send()` (*anyio.streams.stapled.StapledObjectStream* method), 43

`send()` (*anyio.streams.text.TextSendStream* method), 45

`send()` (*anyio.streams.text.TextStream* method), 46

`send()` (*anyio.streams.tls.TLSStream* method), 47

`send_eof()` (*anyio.abc.ByteStream* method), 35

`send_eof()` (*anyio.abc.ObjectStream* method), 34

`send_eof()` (*anyio.streams.stapled.StapledByteStream* method), 43

`send_eof()` (*anyio.streams.stapled.StapledObjectStream* method), 44

`send_eof()` (*anyio.streams.text.TextStream* method), 46

`send_eof()` (*anyio.streams.tls.TLSStream* method), 47

`send_nowait()` (*anyio.streams.memory.MemoryObjectSendStream* method), 41

`send_signal()` (*anyio.abc.Process* method), 55

`sendto()` (*anyio.abc.UDPSocket* method), 53

`serve()` (*anyio.abc.Listener* method), 35

`serve()` (*anyio.abc.SocketListener* method), 53

`serve()` (*anyio.streams.stapled.MultiListener* method), 42

`serve()` (*anyio.streams.tls.TLSListener* method), 48

`server_side` (*anyio.streams.tls.TLSAttribute* attribute), 46

`set()` (*anyio.abc.Event* method), 57

`set_total_tokens()` (*anyio.abc.CapacityLimiter* method), 58

`shared_ciphers` (*anyio.streams.tls.TLSAttribute* attribute), 46

`shield()` (*anyio.abc.CancelScope* property), 29

`sleep()` (in module *anyio*), 27

`sleep_until_stopped()` (*anyio.abc.BlockingPortal* method), 31

SocketAttribute (class in *anyio.abc*), 52

SocketListener (class in *anyio.abc*), 53

SocketStream (class in *anyio.abc*), 53

`spawn()` (*anyio.abc.TaskGroup* method), 29

`spawn_task()` (*anyio.abc.BlockingPortal* method), 31

`ssl_object` (*anyio.streams.tls.TLSAttribute* attribute), 46

`standard_compatible` (*anyio.streams.tls.TLSAttribute* attribute), 46

StapledByteStream (class in *anyio.streams.stapled*), 42

StapledObjectStream (class in *anyio.streams.stapled*), 43

`start_blocking_portal()` (in module *anyio*), 30
`stderr()` (*anyio.abc.Process* property), 55
`stdin()` (*anyio.abc.Process* property), 55
`stdout()` (*anyio.abc.Process* property), 55
`stop()` (*anyio.abc.BlockingPortal* method), 31
`stop_from_external_thread()`
 (*anyio.abc.BlockingPortal* method), 31

T

`TaskGroup` (class in *anyio.abc*), 29
`TaskInfo` (class in *anyio*), 59
`terminate()` (*anyio.abc.Process* method), 55
`TextReceiveStream` (class in *anyio.streams.text*), 44
`TextSendStream` (class in *anyio.streams.text*), 44
`TextStream` (class in *anyio.streams.text*), 45
`tls_version` (*anyio.streams.tls.TLSAttribute* attribute), 47
`TLSAttribute` (class in *anyio.streams.tls*), 46
`TLSListener` (class in *anyio.streams.tls*), 48
`TLSSStream` (class in *anyio.streams.tls*), 47
`total_tokens()` (*anyio.abc.CapacityLimiter* property), 58
`typed_attribute()` (in module *anyio*), 28
`TypedAttributeLookupError`, 60
`TypedAttributeProvider` (class in *anyio*), 28
`TypedAttributeSet` (class in *anyio*), 28

U

`UDPSocket` (class in *anyio.abc*), 53
`UnreliableObjectReceiveStream` (class in *anyio.abc*), 33
`UnreliableObjectSendStream` (class in *anyio.abc*), 33
`UnreliableObjectStream` (class in *anyio.abc*), 34
`unwrap()` (*anyio.streams.tls.TLSSStream* method), 47

V

`value()` (*anyio.abc.Semaphore* property), 56

W

`wait()` (*anyio.abc.Condition* method), 57
`wait()` (*anyio.abc.Event* method), 57
`wait()` (*anyio.abc.Process* method), 55
`wait_all_tasks_blocked()` (in module *anyio*), 60
`wait_socket_readable()` (in module *anyio*), 52
`wait_socket_writable()` (in module *anyio*), 52
`WouldBlock`, 60
`wrap()` (*anyio.streams.tls.TLSSStream* class method), 47
`wrap_async_context_manager()`
 (*anyio.abc.BlockingPortal* method), 31
`wrapped()` (*anyio.AsyncFile* property), 33