AnyIO

Release 1.0.0

Alex Grönholm

May 14, 2019

CONTENTS

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
- Synchronization primitives (locks, conditions, events, semaphores, queues)
- 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.

CHAPTER

ONE

THE MANUAL

1.1 The basics

AnyIO requires Python 3.5.3 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 *`anyio.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 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


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, `TaskGroupError` 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 a `TaskGroupError`.

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


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. Shieldin 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. Events can also be reused by clearing the triggered state.

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)
```

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)
```

(continues on next page)

(continued from previous page)

```
        async with condition:
            await condition.notify_all()

run(main)
```

1.4.5 Queues

Queues are used to send objects between tasks. Queues have two central concepts:

- Producers add things to the queue
- Consumers take things from the queue

When an item is inserted into the queue, it will be given to the next consumer that tries to get an item from the queue. Each item is only ever given to a single consumer.

Queues have a maximum capacity which is determined on creation and cannot be changed later. When the queue is full, any attempt to put an item to it will block until a consumer retrieves an item from the queue. If you wish to avoid blocking on either operation, you can use the *full()* and *empty()* methods to find out about either condition.

Example:

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


async def produce(queue):
    for number in range(10):
        await queue.put(number)
        await sleep(1)


async def main():
    queue = create_queue(100)
    async with create_task_group() as tg:
        await tg.spawn(produce, queue)
        while True:
            number = await queue.get()
            print(number)
            if number == 9:
                break

run(main)
```

1.5 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. To 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.

1.5.1 Running a function in a worker thread

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

```
import time

from anyio import run_in_thread, run


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

run(main)
```

1.5.2 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_in_thread, run


def blocking_function():
    run_async_from_thread(sleep, 5)


async def main():
    await run_in_thread(blocking_function)

run(main)
```

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

1.6 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 aopen, run


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

(continues on next page)

(continued from previous page)

```
        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 aopen, run


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

run(main)
```

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, with TLS encryption support)
- UNIX domain sockets (client + server)
- UDP sockets

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

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_all(b'Client\n')
        response = await client.receive_until(b'\n', 1024)
        print(response)

run(main)
```

To receive incoming TCP connections, you first create a TCP server with *anyio.create_tcp_server()* and then asynchronously iterate over *accept_connections()* and then hand off the yielded client streams to their dedicated tasks:

```
from anyio import create_task_group, create_tcp_server, run


async def serve(client):
    async with client:
        name = await client.receive_until(b'\n', 1024)
        await client.send_all(b'Hello, %s\n' % name)


async def main():
    async with create_task_group() as tg, await create_tcp_server(1234) as server:
        async for client in server.accept_connections():
            await tg.spawn(serve, client)

run(main)
```

The `async for` loop will automatically exit when the server is closed.

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_all(b'Client\n')
        response = await client.receive_until(b'\n', 1024)
        print(response)

run(main)
```

And the server:

```
from anyio import create_task_group, create_unix_server, run


async def serve(client):
    async with client:
        name = await client.receive_until(b'\n', 1024)
        await client.send_all(b'Hello, %s\n' % name)


async def main():
    async with create_task_group() as tg, await create_unix_server('/tmp/mysock') as␣
↪server:
        async for client in server.accept_connections():
            await tg.spawn(serve, client)

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:

```
from anyio import create_udp_socket, run


async def main():
    async with await create_udp_socket(port=1234) as socket:
        async for packet, (host, port) in socket.receive_packets(1024):
            await socket.send(b'Hello, ' + packet, host, port)

run(main)
```

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_udp_socket, run


async def main():
    async with await create_udp_socket(target_host='hostname', target_port=1234) as␣
↪socket:
        await socket.send(b'Hi there!\n')

run(main)
```

1.7.4 Working with TLS

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_task_group, create_tcp_server, run


async def serve(client):
    async with client:
        name = await client.receive_until(b'\n', 1024)
        await client.send_all(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')

    async with create_task_group() as tg:
        async with await create_tcp_server(1234, ssl_context=context) as server:
            async for client in server.accept_connections():
                await tg.spawn(serve, client)

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)
```

(continues on next page)

(continued from previous page)

```
    context.load_verify_locations(cafile='cert.pem')

    async with await connect_tcp('localhost', 1234, ssl_context=context, autostart_
↪tls=True) as client:
        await client.send_all(b'Client\n')
        response = await client.receive_until(b'\n', 1024)
        print(response)

run(main)
```

Manually establishing TLS

Some protocols, like FTP or IMAP, support a technique called "opportunistic TLS". This means that if the server advertises the capability of establishing a secure connection, the client can initiate a TLS handshake after notifying the server using a protocol specific manner.

To do this, you want to prevent the automatic TLS handshake on the server by passing the `autostart_tls=False` option:

```
import ssl

from anyio import create_task_group, create_tcp_server, finalize, run


async def serve(client):
    async with client, finalize(client.receive_delimited_chunks(b'\n', 100)) as lines:
        async for line in lines:
            print('Received "{}"'.format(line.decode('utf-8')))
            if line == b'STARTTLS':
                await client.start_tls()
            elif line == b'QUIT':
                return


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')

    async with create_task_group() as tg:
        async with await create_tcp_server(1234, ssl_context=context, autostart_
↪tls=False) as server:
            async for client in server.accept_connections():
                await tg.spawn(serve, client)

run(main)
```

On the client, you will need to omit the `autostart_tls` option:

```
import ssl

from anyio import connect_tcp, run

```

(continues on next page)

(continued from previous page)

```
async def main():
    # Skip these unless connecting to a server with a self signed certificate
    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_all(b'DUMMY\n')
        await client.send_all(b'STARTTLS\n')
        await client.start_tls()

        # From this point on, all communication is encrypted
        await client.send_all(b'ENCRYPTED\n')
        await client.send_all(b'QUIT\n')

run(main)
```

Dealing with ragged EOFs

According to the TLS standard, encrypted connections should end with a shutdown 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 shutdown handshake, you need to pass the `tls_standard_compatible=False` option to `connect_tcp()` or `create_tcp_server()` (depending on whether you're implementing a client or a server, obviously).

1.8 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 receive_signals, run


async def main():
    async with receive_signals(signal.SIGTERM, signal.SIGHUP) as signals:
        async for signum in signals:
            if signum == signal.SIGTERM:
                return
            elif signum == signal.SIGHUP:
                print('Reloading configuration')

run(main)
```

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

1.9 Testing with AnyIO

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

1.9.1 Creating asynchronous tests

To mark a coroutine function to be run via *anyio.run()*, simply add the `@pytest.mark.anyio` decorator:

```
import pytest


@pytest.mark.anyio
async def test_something():
    pass
```

1.9.2 Asynchronous fixtures

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

```
import pytest


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


@pytest.mark.anyio
async def test_server(server):
    result = await server.do_something()
    assert result == 'foo'
```

Any coroutine fixture that is activated by a test marked with `@pytest.mark.anyio` will be run with the same backend as the test itself. Both plain coroutine functions and asynchronous generator functions are supported in the same manner as pytest itself does with regular functions and generator functions.

Note: If you need Python 3.5 compatibility, please use the async_generator library to replace the async generator syntax that was introduced in Python 3.6.

1.9.3 Specifying the backend to run on

By default, all tests are run against the default backend (asyncio). The pytest plugin provides a command line switch (`--anyio-backends`) for selecting which backend(s) to run your tests against. By specifying a special value, `all`, it will run against all available backends.

For example, to run your test suite against the curio and trio backends:

```
pytest --anyio-backends=curio,trio
```

1.10 API reference

1.10.1 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

Return type +T_Retval

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

1.10.2 Miscellaneous

anyio.**finalize**(*resource*)
Return a context manager that automatically closes an asynchronous resource on exit.

Parameters **resource** (*~T_Agen*) – an asynchronous generator or other resource with an aclose() method

Return type Asynccontextmanager[~T_Agen]

Returns an asynchronous context manager that yields the given object

coroutine anyio.**sleep**(*delay*)
Pause the current task for the specified duration.

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

Return type Coroutine[Any, Any, None]

anyio.**get_cancelled_exc_class**()
Return the current async library's cancellation exception class.

Return type Type[BaseException]

1.10.3 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

Return type Asynccontextmanager[*CancelScope*]

Returns an asynchronous context manager that yields a cancel scope

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

Return type Asynccontextmanager[*CancelScope*]

Returns an asynchronous context manager that yields a cancel scope

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

coroutine 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

coroutine anyio.**current_time**()
Return the current value of the event loop's internal clock.

:return the clock value (seconds) :rtype: float

class anyio.abc.**CancelScope**

coroutine **cancel**()
Cancel this scope immediately.

cancel_called
True if *cancel()* has been called.

Return type bool

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

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

Return type float

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

1.10.4 Task groups

anyio.**create_task_group**()
Create a task group.

Return type *TaskGroup*

Returns a task group

class anyio.abc.**TaskGroup**
Groups several asynchronous tasks together.

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

coroutine **spawn**(*self*, *func*, **args*, *name=None*)
Launch a new task in this task group.

Parameters

- **func** (Callable[..., Coroutine[+T_co, -T_contra, +V_co]]) – 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

1.10.5 Threads

coroutine anyio.**run_in_thread**(*func*, **args*)
Start a thread that calls the given function with the given arguments.

Parameters

- **func** (Callable[..., +T_Retval]) – a callable
- **args** – positional arguments for the callable

Return type Awaitable[+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

1.10.6 Async file I/O

coroutine anyio.**aopen**(*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()`.

Returns an asynchronous file object

Return type *AsyncFile*

class anyio.abc.**AsyncFile**

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
- close

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 aopen(...) as f:
    async for line in f:
        print(line)
```

1.10.7 Sockets and networking

coroutine anyio.**connect_tcp**(*address*, *port*, ***, *ssl_context=None*, *autostart_tls=False*, *bind_host=None*, *bind_port=None*, *tls_standard_compatible=True*)

Connect to a host using the TCP protocol.

Parameters

- **address** (Union[str, IPv4Address, IPv6Address]) – the IP address or host name to connect to
- **port** (int) – port on the target host to connect to
- **ssl_context** (Optional[SSLContext]) – default SSL context to use for TLS handshakes
- **autostart_tls** (bool) – True to do a TLS handshake on connect
- **bind_host** (Union[str, IPv4Address, IPv6Address, None]) – the interface address or name to bind the socket to before connecting
- **bind_port** (Optional[int]) – the port to bind the socket to before connecting
- **tls_standard_compatible** (bool) – 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.

Return type *SocketStream*

Returns a socket stream object

coroutine 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

coroutine anyio.**create_tcp_server**(*port=0*, *interface=None*, *ssl_context=None*, *autostart_tls=True*, *tls_standard_compatible=True*)

Start a TCP socket server.

Parameters

- **port** (int) – port number to listen on
- **interface** (Union[str, IPv4Address, IPv6Address, None]) – interface to listen on (if omitted, listen on any interface)
- **ssl_context** (Optional[SSLContext]) – an SSL context object for TLS negotiation
- **autostart_tls** (bool) – automatically do the TLS handshake on new connections if ssl_context has been provided
- **tls_standard_compatible** (bool) – If True, performs the TLS shutdown handshake before closing a connected stream and requires that the client does this as well. Otherwise, SSLEOFError may be raised during reads from a client stream. Some protocols, such as HTTP, require this option to be False. See wrap_socket() for details.

Return type *SocketStreamServer*

Returns a server object

coroutine anyio.**create_unix_server**(*path*, *, *mode=None*)

Start a UNIX socket server.

Not available on Windows.

Parameters

- **path** (Union[str, PathLike]) – path of the socket
- **mode** (Optional[int]) – permissions to set on the socket

Return type *SocketStreamServer*

Returns a server object

coroutine anyio.**create_udp_socket**(*, *interface=None*, *port=None*, *target_host=None*, *target_port=None*)

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

- **interface** (Union[str, IPv4Address, IPv6Address, None]) – interface to bind to
- **port** (Optional[int]) – port to bind to
- **target_host** (Union[str, IPv4Address, IPv6Address, None]) – remote host to set as the default target
- **target_port** (Optional[int]) – port on the remote host to set as the default target

Return type *UDPSocket*

Returns a UDP socket

coroutine anyio.**wait_socket_readable**(*sock*)

Wait until the given socket has data to be read.

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

Raises

- **anyio.exceptions.ClosedResourceError** – if the socket was closed while waiting for the socket to become readable
- **anyio.exceptions.ResourceBusyError** – if another task is already waiting for the socket to become readable

Return type Awaitable[None]

coroutine anyio.**wait_socket_writable**(*sock*)

Wait until the given socket can be written to.

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

Raises

- **anyio.exceptions.ClosedResourceError** – if the socket was closed while waiting for the socket to become writable
- **anyio.exceptions.ResourceBusyError** – if another task is already waiting for the socket to become writable

Return type Awaitable[None]

coroutine anyio.**notify_socket_close**(*sock*)

Notify any relevant tasks that you are about to close a socket.

This will cause `ClosedResourceError` to be raised on any task waiting for the socket to become readable or writable.

Parameters **sock** (socket) – the socket to be closed after this

Return type Awaitable[None]

class anyio.abc.**Stream**

buffered_data
Return the data currently in the read buffer.

Return type bytes

coroutine close(*self*)
Close the stream.

Return type None

receive_chunks(*max_size*)
Return an async iterable which yields chunks of bytes as soon as they are received.

The generator will yield new chunks until the stream is closed.

Parameters **max_size** (int) – maximum number of bytes to return in one iteration

Return type Asynciterable[bytes]

Returns an async iterable yielding bytes

Raises **ssl.SSLEOFError** – if tls_standard_compatible was set to True in a TLS stream and the peer prematurely closed the connection

receive_delimited_chunks(*delimiter*, *max_chunk_size*)
Return an async iterable which yields chunks of bytes as soon as they are received.

The generator will yield new chunks until the stream is closed.

Parameters

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

Return type Asynciterable[bytes]

Returns an async iterable yielding bytes

Raises

- **anyio.exceptions.IncompleteRead** – if the stream was closed before the delimiter was found
- **anyio.exceptions.DelimiterNotFound** – if the delimiter is not found within the bytes read up to the maximum allowed
- **ssl.SSLEOFError** – if tls_standard_compatible was set to True in a TLS stream and the peer prematurely closed the connection

coroutine receive_exactly(*self*, *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

- **anyio.exceptions.IncompleteRead** – if the stream was closed before the requested amount of bytes could be read from the stream
- **ssl.SSLEOFError** – if `tls_standard_compatible` was set to `True` in a TLS stream and the peer prematurely closed the connection

coroutine receive_some(*self*, *max_bytes*)
Reads up to the given amount of bytes from the stream.

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

Return type `bytes`

Returns the bytes read

Raises **ssl.SSLEOFError** – if `tls_standard_compatible` was set to `True` in a TLS stream and the peer prematurely closed the connection

coroutine receive_until(*self*, *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, including the delimiter

Raises

- **anyio.exceptions.IncompleteRead** – if the stream was closed before the delimiter was found
- **anyio.exceptions.DelimiterNotFound** – if the delimiter is not found within the bytes read up to the maximum allowed
- **ssl.SSLEOFError** – if `tls_standard_compatible` was set to `True` in a TLS stream and the peer prematurely closed the connection

coroutine send_all(*self*, *data*)
Send all of the given data to the other end.

Parameters **data** (`bytes`) – the bytes to send

Return type `None`

class anyio.abc.**SocketStream**
Bases: *anyio.abc.Stream*

alpn_protocol
The ALPN protocol selected during the TLS handshake.

Return type `Optional[str]`

Returns The selected ALPN protocol, or `None` if no ALPN protocol was selected

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

cipher
The cipher selected in the TLS handshake.

See `ssl.SSLSocket.cipher()` for more information.

Return type Tuple[str, str, int]

Returns a 3-tuple of (cipher name, TLS version which defined it, number of bits)

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

get_channel_binding(*cb_type='tls-unique'*)
Get the channel binding data for the current connection.

See ssl.SSLSocket.get_channel_binding() for more information.

Parameters **cb_type** (str) – type of the channel binding to get

Return type bytes

Returns the channel binding data

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

getpeercert(*binary_form=False*)
Get the certificate for the peer on the other end of the connection.

See ssl.SSLSocket.getpeercert() for more information.

Parameters **binary_form** (bool) – False to return the certificate as a dict, True to return it as bytes

Return type Union[Dict[str, Union[str, tuple]], bytes, None]

Returns the peer's certificate, or None if there is not certificate for the peer

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

getsockopt(*level*, *optname*, **args*)
Get a socket option from the underlying socket.

Returns the return value of getsockopt()

server_hostname
The server host name.

Return type Optional[str]

Returns the server host name, or None if this is the server side of the connection

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

server_side
True if this is the server side of the connection, False if this is the client.

Return type bool

Returns True if this is the server side, False if this is the client side

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

setsockopt(*level*, *optname*, *value*, **args*)
Set a socket option.

This calls setsockopt() on the underlying socket.

Return type None

shared_ciphers
The list of ciphers supported by both parties in the TLS handshake.

See ssl.SSLSocket.shared_ciphers() for more information.

Return type List[Tuple[str, str, int]]

Returns a list of 3-tuples (cipher name, TLS version which defined it, number of bits)

Raises **anyio.exceptions.TLSRequired** – if a TLS handshake has not been done

coroutine start_tls(*self*, *context=None*)
Start the TLS handshake.

If the handshake fails, the stream will be closed.

Parameters **context** (Optional[SSLContext]) – an explicit SSL context to use for the handshake

Return type None

tls_version
The TLS version negotiated during the TLS handshake.

See ssl.SSLSocket.version() for more information.

Return type Optional[str]

Returns the TLS version string (e.g. "TLSv1.3"), or None if the underlying socket is not using TLS

class anyio.abc.**SocketStreamServer**

coroutine accept(*self*)
Accept an incoming connection.

Return type *SocketStream*

Returns the socket stream for the accepted connection

accept_connections()
Return an async iterable yielding streams from accepted incoming connections.

Return type Asynciterable[*SocketStream*]

Returns an async context manager

address
Return the bound address of the underlying socket.

Return type Union[Tuple[str, int], Tuple[str, int, int, int], str]

coroutine close(*self*)
Close the underlying socket.

Return type None

getsockopt(*level*, *optname*, **args*)
Get a socket option from the underlying socket.

Returns the return value of getsockopt()

port
The currently bound port of the underlying TCP socket.

Equivalent to server.address[1]. :raises ValueError: if the socket is not a TCP socket

Return type int

setsockopt(*level*, *optname*, *value*, **args*)
Set a socket option.

This calls setsockopt() on the underlying socket.

Return type None

class anyio.abc.**UDPSocket**

address
Return the bound address of the underlying socket.

Return type Union[Tuple[str, int], Tuple[str, int, int, int]]

coroutine close(*self*)
Close the underlying socket.

Return type None

getsockopt(*level*, *optname*, **args*)
Get a socket option from the underlying socket.

Returns the return value of getsockopt()

port
Return the currently bound port of the underlying socket.

Equivalent to socket.address[1].

Return type int

coroutine receive(*self*, *max_bytes*)
Receive a datagram.

No more than max_bytes of the received datagram will be returned, even if the datagram was really larger.

Parameters **max_bytes** (int) – maximum amount of bytes to be returned

Return type Tuple[bytes, str]

Returns the bytes received

receive_packets(*max_size*)
Return an async iterable which yields packets read from the socket.

The iterable exits if the socket is closed.

Return type Asynciterable[Tuple[bytes, str]]

Returns an async iterable yielding (bytes, source address) tuples

coroutine send(*self*, *data*, *address=None*, *port=None*)
Send a datagram.

If the default destination has been set, then address and port are optional.

Parameters

- **data** (bytes) – the bytes to send
- **address** (Optional[str]) – the destination IP address or host name
- **port** (Optional[int]) – the destination port

Return type None

setsockopt(*level*, *optname*, *value*, **args*)
Set a socket option.

This calls setsockopt() on the underlying socket.

Return type None

1.10.8 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**()
Create an asynchronous condition.

Return type *Condition*

Returns a condition object

anyio.**create_queue**(*capacity*)
Create an asynchronous queue.

Parameters **capacity** (int) – maximum number of items the queue will be able to store

Return type *Queue*

Returns a queue object

class anyio.abc.**Semaphore**

value
The current value of the semaphore.

Return type int

class anyio.abc.**Lock**

locked()
Return True if the lock is currently held.

Return type bool

class anyio.abc.**Event**

clear()
Clear the flag, so that listeners can receive another notification.

Return type None

is_set()
Return `True` if the flag is set, `False` if not.

Return type `None`

coroutine set(*self*)
Set the flag, notifying all listeners.

Return type `None`

coroutine wait(*self*)
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**

locked()
Return True if the lock is set.

Return type `bool`

coroutine notify(*self*, *n=1*)
Notify exactly n listeners.

Return type `None`

coroutine notify_all(*self*)
Notify all the listeners.

Return type `None`

coroutine wait(*self*)
Wait for a notification.

Return type `None`

class anyio.abc.**Queue**

empty()
Return `True` if the queue is not holding any items.

Return type `bool`

full()
Return `True` if the queue is holding the maximum number of items.

Return type `bool`

coroutine get()
Get an item from the queue.

If there are no items in the queue, this method will block until one is available.

Returns the removed item

coroutine put(*self*, *item*)
Put an item into the queue.

If the queue is currently full, this method will block until there is at least one free slot available.

Parameters **item** – the object to put into the queue

Return type None

qsize()
Return the number of items the queue is currently holding.

Return type int

1.10.9 Operating system signals

coroutine anyio.**receive_signals**(**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.

1.10.10 Testing and debugging

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

coroutine anyio.**get_current_task**()
Return the current task.

Return type *TaskInfo*

Returns a representation of the current task

coroutine 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

coroutine anyio.**wait_all_tasks_blocked**()
Wait until all other tasks are waiting for something.

Return type None

1.11 Version history

This library adheres to Semantic Versioning.

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