Designing Libraries for Async and Synchronous I/O

Seth Michael Larson

Creating a library that supports both async and sync APIs is tough but a big win for users. Here's a summary of three tools available now that I've used to help design and maintain libraries with async and sync I/O.

Each tool solves one of two problems:

- **Support both async and sync in the same codebase**
- **Support multiple async libraries (asyncio, Trio)**

If you'd like to view a simple project using all of these tools together I have created one on GitHub.

Unasync

'unasync' is a library that tokenizes your Python code, transforms tokens for async code into their synchronous counterparts, and then re-renders the new synchronous code into a corresponding file.

See the code examples on the right for some of the tokens that unasync transforms:

```
# Classes prefixed with 'AsyncX...' are helpfully
# renamed to 'SyncX...' for simpler imports.
class AsyncClass:          → class SyncClass:

# async/await is handled by removing the
# async and await keywords.
async def f(): ...         → def f(): ...
ret = await f()            → ret = f()

# Async context managers are transformed
# into regular context managers.
async with ctx():          → with ctx():
async def __aenter__():    → def __enter__():
async def __aexit__(*_)    → def __exit__(*_):

# Async iterators are transformed into
# regular iterators.
x.__aiter__()              → x.__iter__()
iter.__anext__()           → iter.__next__()
async for x in y:          → for x in y:
StopAsyncIteration         → StopIteration

# Typing annotations, notice unasync also
# handles forward annotations within strings.
def f() → "AsyncClass":    → def f() → "SyncClass":
typing.AsyncIterator       → typing.Iterator
typing.AsyncGenerator      → typing.Generator
typing.AsyncIterable       → typing.Iterable

# There are some async statements that don't
# have a direct sync counterpart so unasync
# can't do anything with them.
async def __await__(): → ???
```

After adding support for both sync and async the next step is supporting multiple async libraries. The libraries commonly used in the Python community are asyncio, Trio, Twisted, and Curio.

Supporting all of these is a challenge but can be made easier with the following two libraries:

Sniffio

‘sniffio’ is a library that can detect which async library your code is running under. The package can detect asyncio, Trio, and Curio. By detecting which library is running you then know which library-specific APIs can be used safely.

This also means you can lazily import ‘trio’ only when Trio is detected as the current async library. No need to make Trio a direct dependency.

A pattern I’ve found useful is to group all library-specific code into one file each that all have an identical API. That way you can call sniffio’s detection one time and know which set of APIs should be used for the duration of the program.

```python
import sniffio

try:
    # Detect the current async library
    async_lib = sniffio.current_async_library()

    # Lazily-load so users don't need 'trio'
    # installed to use 'asyncio'.
    if async_lib == "asyncio":
        import asyncio
        # <asyncio-specific code>
    elif async_lib == "trio":
        import trio
        # <trio-specific code>
    else:
        raise RuntimeError(
          f"Unsupported async library: {asynclib!r}"
        )
except sniffio.AsyncLibraryNotFoundError:
    raise RuntimeError(
      "Couldn't detect async library"
    ) from None
```

AnyIO

‘anyio’ is a library that provides a single interface that can be used interchangeably from asyncio, Trio, and Curio programs.

For most projects trying to support multiple async libraries this should be your first stop.

The interfaces that are provided include:

- Structured Concurrency Primitives
- Synchronization Primitives
- Networking (TCP / TLS / UDP)
- Asynchronous File I/O
- Signals and Threads

AnyIO also provides its own pytest plugin ‘pytest-anyio’ that makes writing test cases for multiple async libraries a breeze.

AnyIO uses Sniffio under the hood to detect which async library is being used and provide the correct implementation.

You can read AnyIO’s documentation for a list of all features and usages.

```python
import anyio

# Structured Concurrency Primitives

# Task Groups
async with anyio.create_task_group() as group:
    async def f(x):
        await anyio.sleep(x)
    for x in range(10):
        await group.spawn(f, x)

# Timeouts
async with anyio.move_on_after(1):
    # Operation will be cancelled
    # if it takes longer than 1 second.

# Cancellation
async with anyio.open_cancel_scope() as cancel_scope:
    stuff_that_can_be_cancelled()
if cancel_scope.cancel_called:
    # Cancellation handling

# Synchronization Primitives
anyio.create_lock()
anyio.create_queue()
anyio.create_capacity_limiter()
anyio.create_event()
anyio.create_condition()

# Networking: TCP / TLS / UDP
sock = await anyio.connect_tcp()
await sock.start_tls()

sock = await anyio.create_udp_socket()
await sock.send()
await sock.receive()
```