universalasync

Release 0.3.1.1

MrNaif2018

Jan 10, 2024

CONTENTS:

CHAPTER

ONE

API REFERENCE

universalasync.**async_to_sync_wraps**(*function: Callable*) → Callable

Wrap an async method/property to universal method.

This allows to run wrapped methods in both async and sync contexts transparently without any additional code

When run from another thread, it runs coroutines in new thread's event loop

See *Example* for full example

Parameters
function (*Callable*) – function/property to wrap

Returns
Callable – modified function

universalasync.**get_event_loop**() → AbstractEventLoop

Useful utility for getting event loop. Acts like get_event_loop(), but also creates new event loop if needed

This will return a working event loop in 100% of cases.

Returns
asyncio.AbstractEventLoop – event loop

universalasync.**idle**() → None

Useful for making event loop idle in the main thread for other threads to work

universalasync.**wrap**(*source: object*) → object

Convert all public async methods/properties of an object to universal methods.

See *async_to_sync_wraps()* for more info

Parameters
source (*object*) – object to convert

Returns
object – converted object. Note that parameter passed is being modified anyway

A library to help automate the creation of universal python libraries

CHAPTER

TWO

OVERVIEW

Have you ever been frustrated that you need to maintain both sync and async versions of your library, even thought their code differs by just async and await? You might have came up to rewriting your code before release or other unreliable solutions.

This library helps you to focus only on the main async implementation of your library: sync one will be created automatically

Via decorating all your public methods, the wrapped functions automatically detect different conditions and run the functions accordingly.

If user uses your library in async context, minimal overhead is added, it just returns the coroutine right away.

Otherwise the library calls the coroutine via various loop methods, like as if you did it manually.

There should be no issues at all, the only limitation is that signals and async subprocesses are supported only when running in the main thread.

Also note that when run from a different os thread, the library will create a new event loop there and run coroutines.

This means that you might need to adapt your code a bit in case you use some resources bound to a certain event loop (like `aiohttp.ClientSession`).

You can see an example of how this could be solved here

CHAPTER

THREE

INSTALLATION

```
pip install universalasync
```

CHAPTER

FOUR

EXAMPLE OF USAGE

```python
# wrap needed methods one by one
class Client:
    @async_to_sync_wraps
    async def help():
        ...

    @async_to_sync_wraps
    @property
    async def async_property():
        ...

# or wrap whole classes
@wrap
class Client:
    async def help(self):
        ...

    @property
    async def async_property():
        ...

client = Client()

def sync_call():
    client.help()
    client.async_property

async def async_call():
    await client.help()
    await client.async_property

# works in all cases
sync_call()
asyncio.run(async_call())
threading.Thread(target=sync_call).start()
threading.Thread(target=asyncio.run, args=(async_call(),)).start()
```

PYTHON MODULE INDEX

INDEX