Asyncio Module Python 3.x 3.0–3.6

Examples

Coroutine and Delegation Syntax

Before Python 3.5+ was released, the `asyncio` module used generators to mimic asynchronous calls and thus had a different syntax than the current Python 3.5 release.

Python 3.x ≥3.5

Python 3.5 introduced the `async` and `await` keywords. Note the lack of parentheses around the `await func()` call.

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x ≥3.3 , < 3.5

Before Python 3.5, the `@asyncio.coroutine` decorator was used to define a coroutine. The `yield from` expression was used for generator delegation. Note the parentheses around the `yield from func()` .

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff..
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x ≥3.5

Here is an example that shows how two functions can be run asynchronously:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
        print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

Asynchronous Executors

Note: Uses the Python 3.5+ async/await syntax

`asyncio` supports the use of `Executor` objects found in `concurrent.futures` for scheduling tasks asynchronously. Event loops have the function `run_in_executor()` which takes an `Executor` object, a `Callable` , and the Callable's parameters.

Scheduling a task for an `Executor`

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Each event loop also has a "default" `Executor` slot that can be assigned to an `Executor` . To assign an `Executor` and schedule tasks from the loop you use the `set_default_executor()` method.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

There are two main types of `Executor` in `concurrent.futures` , the `ThreadPoolExecutor` and the `ProcessPoolExecutor` . The `ThreadPoolExecutor` contains a pool of threads which can either be manually set to a specific number of threads through the constructor or defaults to the number of cores on the machine times 5. The `ThreadPoolExecutor` uses the pool of threads to execute tasks assigned to it and is generally better at CPU-bound operations rather than I/O bound operations. Contrast that to the `ProcessPoolExecutor` which spawns a new process for each task assigned to it. The `ProcessPoolExecutor` can only take tasks and parameters that are picklable. The most common non-picklable tasks are the methods of objects. If you must schedule an object's method as a task in an `Executor` you must use a `ThreadPoolExecutor` .

Using UVLoop

`uvloop` is an implementation for the `asyncio.AbstractEventLoop` based on libuv (Used by nodejs). It is compliant with 99% of `asyncio` features and is much faster than the traditional `asyncio.EventLoop` . `uvloop` is currently not available on Windows, install it with `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

One can also change the event loop factory by setting the `EventLoopPolicy` to the one in `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

A Simple Websocket

Here we make a simple echo websocket using `asyncio` . We define coroutines for connecting to a server and sending/receiving messages. The communcations of the websocket are run in a `main` coroutine, which is run by an event loop. This example is modified from a prior post .

```
import asyncio
import aiohttp

session = aiohttp.ClientSession()                          # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()
    await echo.connect()
    await echo.send("Hello World!")
    print(await echo.receive())                            # "Hello World!"


if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Synchronization Primitive: Event

Concept

Use an `Event` to **synchronize the scheduling of multiple coroutines** .

Put simply, an event is like the gun shot at a running race: it lets the runners off the starting blocks.

Example

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set()  # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
main_future = asyncio.wait([consumer_a(event),
                            consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event))  # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)
```

Output:

```
Consumer B waiting
Consumer A waiting
EVENT SET
Consumer B triggered
Consumer A triggered
```

Common Misconception about asyncio

Common misconception about asyncio

probably *the* most common misconception about `asnycio` is that it lets you run any task in parallel - sidestepping the GIL (global interpreter lock) and therefore execute blocking jobs in parallel (on separate threads). it does **not** !

`asyncio` (and libraries that are built to collaborate with `asyncio`) build on coroutines: functions that (collaboratively) yield the control flow back to the calling function. note `asyncio.sleep` in the examples above. this is an example of a non-blocking coroutine that waits 'in the background' and gives the control flow back to the calling function (when called with `await`). `time.sleep` is an example of a blocking function. the execution flow of the program will just stop there and only return after `time.sleep` has finished.

a real-live example is the `requests` library which consists (for the time being) on blocking functions only. there is no concurrency if you call any of its functions within `asyncio`. `aiohttp` on the other hand was built with `asyncio` in mind. its coroutines will run concurrently.

- if you have long-running CPU-bound tasks you would like to run in parallel `asyncio` is **not** for you. for that you need `threads` or `multiprocessing`.
- if you have IO-bound jobs running, you *may* run them concurrently using `asyncio`.

Syntax

Parameters

Remarks