Scheduling Calls to Regular Functions

In addition to managing coroutines and I/O callbacks, the `asyncio` event loop can schedule calls to regular functions based on the timer value kept in the loop.

Scheduling a Callback "Soon"

If the timing of the callback does not matter, `call_soon()` can be used to schedule the call for the next iteration of the loop. Any extra positional arguments after the function are passed to the callback when it is invoked. To pass keyword arguments to the callback, use `partial()` from the `functools` module.

```
# asyncio_call_soon.py

import asyncio
import functools


def callback(arg, *, kwarg='default'):
    print('callback invoked with {} and {}'.format(arg, kwarg))


async def main(loop):
    print('registering callbacks')
    loop.call_soon(callback, 1)
    wrapped = functools.partial(callback, kwarg='not default')
    loop.call_soon(wrapped, 2)

    await asyncio.sleep(0.1)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

The callbacks are invoked in the order they are scheduled.

```
$ python3 asyncio_call_soon.py

entering event loop
registering callbacks
callback invoked with 1 and default
callback invoked with 2 and not default
closing event loop
```

Scheduling a Callback with a Delay

To postpone a callback until some time in the future, use `call_later()`. The first argument is the delay in seconds and the second argument is the callback.

```
# asyncio_call_later.py

import asyncio


def callback(n):
    print('callback {} invoked'.format(n))
```

```
async def main(loop):
    print('registering callbacks')
    loop.call_later(0.2, callback, 1)
    loop.call_later(0.1, callback, 2)
    loop.call_soon(callback, 3)

    await asyncio.sleep(0.4)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

In this example, the same callback function is scheduled for several different times with different arguments. The final instance, using `call_soon()`, results in the callback being invoked with the argument 3 before any of the time-scheduled instances, showing that "soon" usually implies a minimal delay.

```
$ python3 asyncio_call_later.py

entering event loop
registering callbacks
callback 3 invoked
callback 2 invoked
callback 1 invoked
closing event loop
```

Scheduling a Callback for a Specific Time

It is also possible to schedule a call to occur at a specific time. The loop uses a monotonic clock, rather than a wall-clock time, to ensure that the value of "now" never regresses. To choose a time for a scheduled callback it is necessary to start from the internal state of that clock using the loop's `time()` method.

```
# asyncio_call_at.py

import asyncio
import time


def callback(n, loop):
    print('callback {} invoked at {}'.format(n, loop.time()))


async def main(loop):
    now = loop.time()
    print('clock time: {}'.format(time.time()))
    print('loop  time: {}'.format(now))

    print('registering callbacks')
    loop.call_at(now + 0.2, callback, 1, loop)
    loop.call_at(now + 0.1, callback, 2, loop)
    loop.call_soon(callback, 3, loop)

    await asyncio.sleep(1)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

Note that the time according to the loop does not match the value returned by `time.time()`.

Note that the time according to the loop does not match the value returned by `time.time()`.

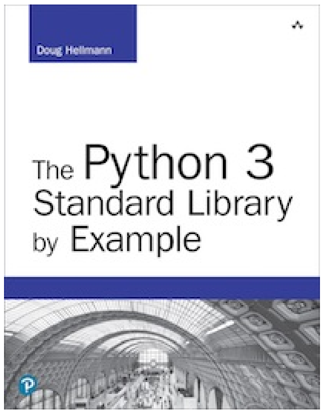
```
$ python3 asyncio_call_at.py

entering event loop
clock time: 1521404411.833459
loop  time: 715855.398664185
registering callbacks
callback 3 invoked at 715855.398744743
callback 2 invoked at 715855.503897727
callback 1 invoked at 715855.601119414
closing event loop
```

Cooperative Multitasking with Coroutines   Producing Results Asynchronously

This page was last updated 2018-03-18.

Navigation
Cooperative Multitasking with Coroutines
Producing Results Asynchronously



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?