Producing Results Asynchronously

A Future represents the result of work that has not been completed yet. The event loop can watch for a Future object's state to indicate that it is done, allowing one part of an application to wait for another part to finish some work.

Waiting for a Future

A Future acts like a coroutine, so any techniques useful for waiting for a coroutine can also be used to wait for the future to be marked done. This example passes the future to the event loop's `run_until_complete()` method.

```
# asyncio_future_event_loop.py

import asyncio


def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)


event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()

    print('scheduling mark_done')
    event_loop.call_soon(mark_done, all_done, 'the result')

    print('entering event loop')
    result = event_loop.run_until_complete(all_done)
    print('returned result: {!r}'.format(result))
finally:
    print('closing event loop')
    event_loop.close()

print('future result: {!r}'.format(all_done.result()))
```

The state of the Future changes to done when `set_result()` is called, and the Future instance retains the result given to the method for retrieval later.

```
$ python3 asyncio_future_event_loop.py

scheduling mark_done
entering event loop
setting future result to 'the result'
returned result: 'the result'
closing event loop
future result: 'the result'
```

A Future can also be used with the `await` keyword, as in this example.

```
# asyncio_future_await.py

import asyncio


def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)


async def main(loop):
```

```
    all_done = asyncio.Future()

    print('scheduling mark_done')
    loop.call_soon(mark_done, all_done, 'the result')

    result = await all_done
    print('returned result: {!r}'.format(result))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

The result of the Future is returned by await, so it is frequently possible to have the same code work with a regular coroutine and a Future instance.

```
$ python3 asyncio_future_await.py

scheduling mark_done
setting future result to 'the result'
returned result: 'the result'
```

Future Callbacks

In addition to working like a coroutine, a Future can invoke callbacks when it is completed. Callbacks are invoked in the order they are registered.

```
# asyncio_future_callback.py

import asyncio
import functools


def callback(future, n):
    print('{}: future done: {}'.format(n, future.result()))


async def register_callbacks(all_done):
    print('registering callbacks on future')
    all_done.add_done_callback(functools.partial(callback, n=1))
    all_done.add_done_callback(functools.partial(callback, n=2))


async def main(all_done):
    await register_callbacks(all_done)
    print('setting result of future')
    all_done.set_result('the result')


event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()
    event_loop.run_until_complete(main(all_done))
finally:
    event_loop.close()
```

The callback should expect one argument, the Future instance. To pass additional arguments to the callbacks, use `functools.partial()` to create a wrapper.

```
$ python3 asyncio_future_callback.py

registering callbacks on future
setting result of future
1: future done: the result
2: future done: the result
```

Scheduling Calls to Regular Functions Executing Tasks Concurrently

Scheduling Calls to Regular Functions
Executing Tasks Concurrently

Quick Links

This page was last updated 2016-12-18.

Navigation

Scheduling Calls to Regular Functions
Executing Tasks Concurrently

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?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example