Composing Coroutines with Control Structures

Linear control flow between a series of coroutines is easy to manage with the built-in language keyword `await`. More complicated structures allowing one coroutine to wait for several others to complete in parallel are also possible using tools in `asyncio`.

Waiting for Multiple Coroutines

It is often useful to divide one operation into many parts and execute them separately. For example, downloading several remote resources or querying remote APIs. In situations where the order of execution doesn't matter, and where there may be an arbitrary number of operations, `wait()` can be used to pause one coroutine until the other background operations complete.

```
# asyncio_wait.py

import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.1 * i)
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    completed, pending = await asyncio.wait(phases)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

Internally, `wait()` uses a set to hold the Task instances it creates. This results in them starting, and finishing, in an unpredictable order. The return value from `wait()` is a tuple containing two sets holding the finished and pending tasks.

```
$ python3 asyncio_wait.py

starting main
waiting for phases to complete
in phase 0
in phase 1
in phase 2
done with phase 0
done with phase 1
done with phase 2
results: ['phase 1 result', 'phase 0 result', 'phase 2 result']
```

There will only be pending operations left if `wait()` is used with a timeout value.

```
# asyncio_wait_timeout.py
```

```
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    try:
        await asyncio.sleep(0.1 * i)
    except asyncio.CancelledError:
        print('phase {} canceled'.format(i))
        raise
    else:
        print('done with phase {}'.format(i))
        return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting 0.1 for phases to complete')
    completed, pending = await asyncio.wait(phases, timeout=0.1)
    print('{} completed and {} pending'.format(
        len(completed), len(pending),
    ))
    # Cancel remaining tasks so they do not generate errors
    # as we exit without finishing them.
    if pending:
        print('canceling tasks')
        for t in pending:
            t.cancel()
    print('exiting main')


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

Those remaining background operations should either be cancelled or finished by waiting for them. Leaving them pending while the event loop continues will let them execute further, which may not be desirable if the overall operation is considered aborted. Leaving them pending at the end of the process will result in warnings being reported.

```
$ python3 asyncio_wait_timeout.py

starting main
waiting 0.1 for phases to complete
in phase 1
in phase 0
in phase 2
done with phase 0
1 completed and 2 pending
cancelling tasks
exiting main
phase 1 cancelled
phase 2 cancelled
```

Gathering Results from Coroutines

If the background phases are well-defined, and only the results of those phases matter, then `gather()` may be more useful for waiting for multiple operations.

```
# asyncio_gather.py

import asyncio
```

```
async def phase1():
    print('in phase1')
    await asyncio.sleep(2)
    print('done with phase1')
    return 'phase1 result'


async def phase2():
    print('in phase2')
    await asyncio.sleep(1)
    print('done with phase2')
    return 'phase2 result'


async def main():
    print('starting main')
    print('waiting for phases to complete')
    results = await asyncio.gather(
        phase1(),
        phase2(),
    )
    print('results: {!r}'.format(results))


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

The tasks created by gather are not exposed, so they cannot be cancelled. The return value is a list of results in the same order as the arguments passed to `gather()`, regardless of the order the background operations actually completed.

```
$ python3 asyncio_gather.py

starting main
waiting for phases to complete
in phase2
in phase1
done with phase2
done with phase1
results: ['phase1 result', 'phase2 result']
```

Handling Background Operations as They Finish

`as_completed()` is a generator that manages the execution of a list of coroutines given to it and produces their results one at a time as they finish running. As with `wait()`, order is not guaranteed by `as_completed()`, but it is not necessary to wait for all of the background operations to complete before taking other action.

```
# asyncio_as_completed.py
```

```
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.5 - (0.1 * i))
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    results = []
```

```
    for next_to_complete in asyncio.as_completed(phases):
        answer = await next_to_complete
        print('received answer {!r}'.format(answer))
        results.append(answer)
    print('results: {!r}'.format(results))
    return results


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

This example starts several background phases that finish in the reverse order from which they start. As the generator is consumed, the loop waits for the result of the coroutine using `await`.

```
$ python3 asyncio_as_completed.py

starting main
waiting for phases to complete
in phase 0
in phase 2
in phase 1
done with phase 2
received answer 'phase 2 result'
done with phase 1
received answer 'phase 1 result'
done with phase 0
received answer 'phase 0 result'
results: ['phase 2 result', 'phase 1 result', 'phase 0 result']
```

Executing Tasks Concurrently

Synchronization Primitives

Quick Links

Waiting for Multiple Coroutines
Gathering Results from Coroutines
Handling Background Operations as They Finish

This page was last updated 2016-12-18.

Navigation

Executing Tasks Concurrently
Synchronization Primitives

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?