

O Concurrency with Processes, Threads, and Coroutines

asyncio — Asynchronous I/O, event loop, and concurrency tools

Executing Tasks Concurrently

Tasks are one of the primary ways to interact with the event loop. Tasks wrap coroutines and track when they are complete. Tasks are subclasses of Future, so other coroutines can wait for them and each has a result that can be retrieved after the task completes.

Starting a Task

To start a task, use create_task() to create a Task instance. The resulting task will run as part of the concurrent operations managed by the event loop as long as the loop is running and the coroutine does not return.

```
# asyncio create task.py
import asyncio
async def task func():
    print('in task func')
    return 'the result'
async def main(loop):
    print('creating task')
    task = loop.create task(task func())
    print('waiting for {!r}'.format(task))
    return_value = await task
    print('task completed {!r}'.format(task))
    print('return value: {!r}'.format(return_value))
event loop = asyncio.get event loop()
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

This example waits for the task to return a result before the main() function exits.

```
$ python3 asyncio_create_task.py

creating task
waiting for <Task pending coro=<task_func() running at
asyncio_create_task.py:12>>
in task_func
task completed <Task finished coro=<task_func() done, defined at
asyncio_create_task.py:12> result='the result'>
return value: 'the result'
```

Canceling a Task

By retaining the Task object returned from create_task(), it is possible to cancel the operation of the task before it completes.

```
# asyncio_cancel_task.py
import asyncio

async def task_func():
    print('in task_func')
```

```
return 'the result'
async def main(loop):
    print('creating task')
    task = loop.create task(task func())
    print('canceling task')
    task.cancel()
    print('canceled task {!r}'.format(task))
        await task
    except asyncio.CancelledError:
        print('caught error from canceled task')
    else:
        print('task result: {!r}'.format(task.result()))
event loop = asyncio.get event loop()
try:
    event loop.run until complete(main(event loop))
finally:
    event loop.close()
```

This example creates and then cancels a task before starting the event loop. The result is a CancelledError exception from run_until_complete().

```
$ python3 asyncio_cancel_task.py

creating task
canceling task
canceled task <Task cancelling coro=<task_func() running at
asyncio_cancel_task.py:12>>
caught error from canceled task
```

If a task is canceled while it is waiting for another concurrent operation, the task is notified of its cancellation by having a CancelledError exception raised at the point where it is waiting.

```
# asyncio cancel task2.py
import asyncio
async def task func():
    print('in task func, sleeping')
    try:
        await asyncio.sleep(1)
    except asyncio.CancelledError:
        print('task_func was canceled')
        raise
    return 'the result'
def task canceller(t):
    print('in task canceller')
    t.cancel()
    print('canceled the task')
async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    loop.call soon(task canceller, task)
    try:
        await task
    except asyncio.CancelledError:
        print('main() also sees task as canceled')
```

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

Catching the exception provides an opportunity to clean up work already done, if necessary.

```
$ python3 asyncio_cancel_task2.py
creating task
in task_func, sleeping
in task_canceller
canceled the task
task_func was canceled
main() also sees task as canceled
```

Creating Tasks from Coroutines

The ensure_future() function returns a Task tied to the execution of a coroutine. That Task instance can then be passed to other code, which can wait for it without knowing how the original coroutine was constructed or called.

```
# asyncio ensure future.py
import asyncio
async def wrapped():
    print('wrapped')
    return 'result'
async def inner(task):
    print('inner: starting')
    print('inner: waiting for {!r}'.format(task))
    result = await task
    print('inner: task returned {!r}'.format(result))
async def starter():
    print('starter: creating task')
    task = asyncio.ensure future(wrapped())
    print('starter: waiting for inner')
    await inner(task)
    print('starter: inner returned')
event loop = asyncio.get event loop()
try:
    print('entering event loop')
    result = event loop.run until complete(starter())
finally:
    event_loop.close()
```

Note that the coroutine given to ensure future() is not started until something uses await to allow it to be executed.

```
$ python3 asyncio_ensure_future.py
entering event loop
starter: creating task
starter: waiting for inner
inner: starting
inner: waiting for <Task pending coro=<wrapped() running at
asyncio_ensure_future.py:12>>
wrapped
inner: task returned 'result'
starter: inner returned
```

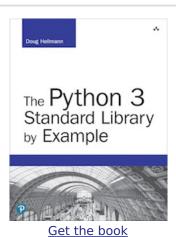
Quick Links

Starting a Task Canceling a Task Creating Tasks from Coroutines

This page was last updated 2016-12-18.

Navigation

Producing Results Asynchronously Composing Coroutines with Control Structures



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 <u>examples for Python 2</u>?

This Site

Module Index

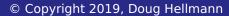
 \boldsymbol{I} Index













Other Writing



