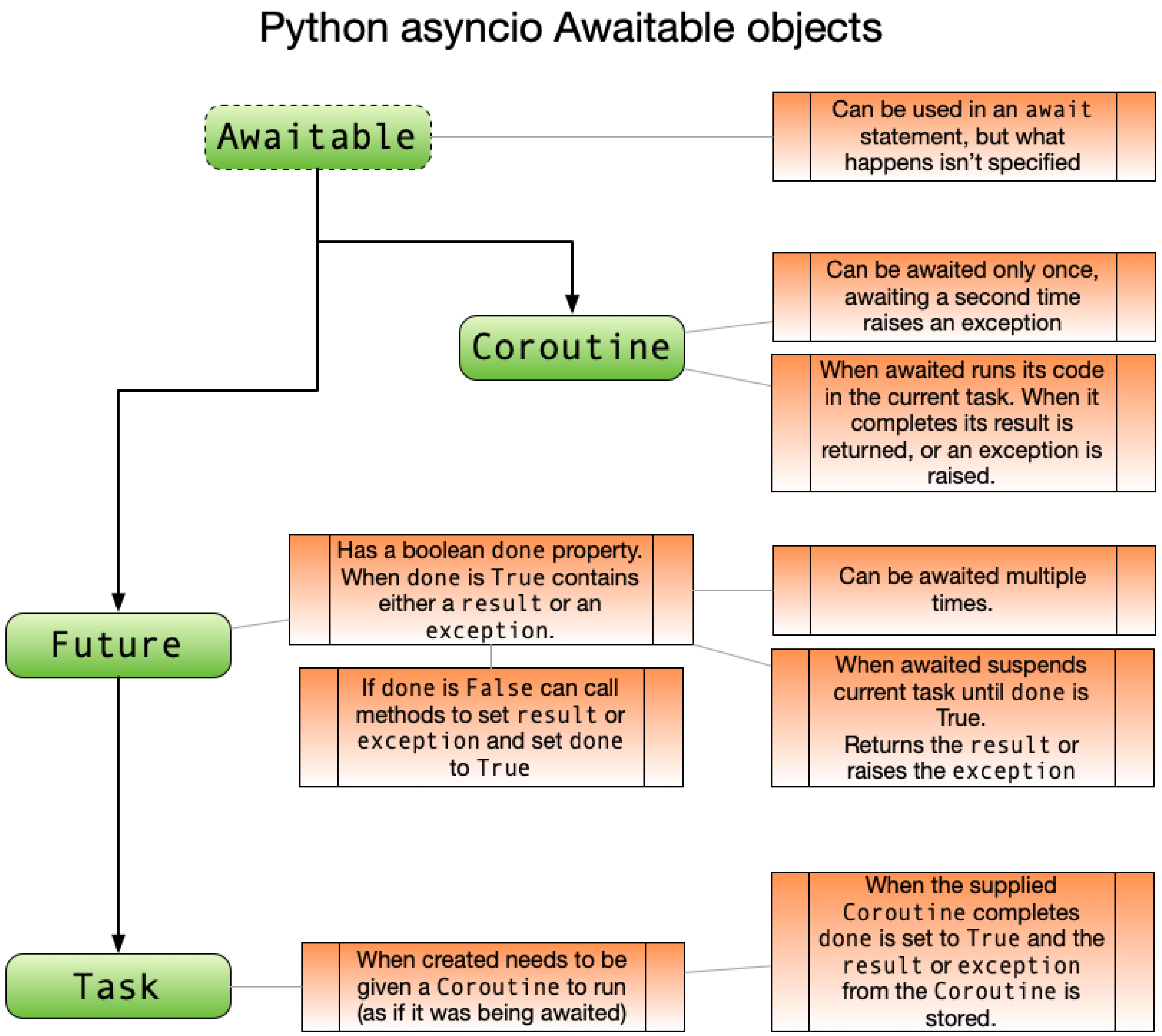
Async python

☰ Tags	Computer Science Python Language and Ecosystem
≡ Topic	Async(coroutines)

Table of contents

Basic concepts of Asyncio



Python asyncio Awaitable objects

Can be used in an `await` statement, but what happens isn't specified

Can be awaited only once, awaiting a second time raises an exception

When awaited runs its code in the current task. When it completes its result is returned, or an exception is raised.

Has a boolean `done` property. When `done` is `True` contains either a `result` or an `exception`.

Can be awaited multiple times.

If `done` is `False` can call methods to set `result` or `exception` and set `done` to `True`

When awaited suspends current task until `done` is True. Returns the `result` or raises the `exception`

When created needs to be given a `Coroutine` to run (as if it was being awaited)

When the supplied `Coroutine` completes `done` is set to `True` and the `result` or `exception` from the `Coroutine` is stored.

So, simplier:

- **Coroutine** is a block of a function with `async/await` However, we can define it in many ways like
  - The code block of asynchronous code inside an `async def` statement.
  - The callable object that the `async def` statement creates.
  - The object of class `Coroutine` that is returned by the callable object when it is called.
- **Task** is a wrapper of a coroutine, so with help of this we can run our coroutines concurently. They are used to schedule and manage the execution of asynchronous

functions in the event loop.

- **Futures** are a special objects that represents the possible state when some external process returns it back. It acts as a placeholder for the eventual outcome of a coroutine and can be used to track its progress or add callbacks. **Basically, this is use only in internals of asyncio or when we try to make new libraries or extend existed ones. Also, it can be used when we're trying to connect blocking libraries with non-blocking libraries.**

> *We need to use tasks, because when we simply call `await coro()` it will block our event loop untils this coro is completed, even though asyncio implicitly creates task*

To show this, there is example:

```
async def delay():
    await asyncio.sleep(4)

async def main():
    t = asyncio.create_task(delay())
    t1 = time.perf_counter()
    await asyncio.sleep(3)
    await t
    print(time.perf_counter() - t1)

asyncio.run(main())
>>> 4.001353383000605
```

And when we simply call:

```
async def main():
    t1 = time.perf_counter()
    await delay()
    await asyncio.sleep(3)
    print(time.perf_counter() - t1)
```

```
asyncio.run(main())
>>> 7.006259050000153
```

More about futures

As it mentioned before, this class is used for representing the result of a computation that may not have been completed yet. They are used to track the state of asynchronous operations and enable callbacks when the operation is done.

```
import asyncio

async def set_future_result(future, result):
    await asyncio.sleep(5)
    future.set_result(result)

async def await_future(future):
    print("I'm waiting for future to set with requests result")
    r = await future
    print(r)

async def main():
    future = asyncio.Future()

    # t1 = asyncio.create_task(await_future(future))
    t2 = asyncio.create_task(set_future_result(future, 'Future

    await await_future(future)

asyncio.run(main())
```

However, we also can do this with help of callbacks and esspecially `add_done_callback()`

```
import asyncio

def on_future_done(future):
    print("Future done. Result:", future.result())

async def set_future_result(future):
    await asyncio.sleep(1)
    future.set_result("Future result")

async def main():
    future = asyncio.Future()
    future.add_done_callback(on_future_done)
    asyncio.create_task(set_future_result(future))

    await asyncio.sleep(2)  # Give enough time for the future t

asyncio.run(main())
```

Although Future objects are used internally by asyncio, you can usually avoid working with them directly by using the `async/await` syntax with coroutines and tasks. Tasks are a subclass of Future objects and provide a more convenient interface for managing the execution of coroutines.

Future objects in asyncio represent the result of a computation that may not have been completed yet. They enable tracking the state of asynchronous operations and allow you to add callbacks when the operation is done.

While Future objects are an essential part of asyncio's internals, you will typically use the `async/await` syntax with coroutines and tasks when writing asyncio code, as it provides a more convenient and intuitive interface.

[DANGEROUS THEME] HOW ALL THIS WORKS UNDER THE HOOD

Event loop

So, let's talk about event loop. Basically, we don't need any of asyncio objects. Event loop just runs scheduled callbacks.
Let's take a look into
`run_once` realization - LINK

This means that in an event loop iteration, the number of `callbacks` being executed is dynamically determined. It does not have fixed time frame, it does not have an fixed number of `callbacks` to run. Everything is dynamically scheduled and thus is very flexible.

Let’s also look at the `run_until_complete`, which `asyncio.run()` uses under the hood - LINK

So, as we understand when we create a task it creates via `create_task` or `ensure_future` (this under the hood calls create_task method when coro is passed). And in `__init__` of a Task class it calls `self ._loop.call_soon( self .__step,  context = self ._context)` which appends task to the _ready list and then just iterates over it with help of heapq.

If any questions go to LINK for more detailed info

Coroutine

So, the type coroutine is now implemented in the python core and has no with asyncio library. However, we can mention `@asyncio.coroutine` decorator. **However, this was removed in python3.10** - LINK

When we tried to run `coroutine` with loop.run_until_complete, we see from the comment that if the argument is a `coroutine` then it would be converted to a Task in the first place, and `loop.run_until_complete` is actually scheduling `Task`s.

I didn’t find Coroutine class definition in source code, so go HERE for it

Future

`Future` has a close relations with `Task`, the defintion HERE

It is linked to the loop, by default utilises `get_event_loop`, hovewer we can pass instance of a loop.

The main method of a `Future` is `set_result()` wich calls `__schedule_callbacks()`:

```
def set_result(self, result):
        """Mark the future done and set its result.

        If the future is already done when this method is calle
        InvalidStateError.
        """
        if self._state != _PENDING:
            raise exceptions.InvalidStateError(f'{self._state}:
        self._result = result
        self._state = _FINISHED
        self.__schedule_callbacks()

def __schedule_callbacks(self):
        """Internal: Ask the event loop to call all callbacks.

        The callbacks are scheduled to be called as soon as poss
        clears the callback list.
        """
        callbacks = self._callbacks[:]
        if not callbacks:
            return

        self._callbacks[:] = []
        for callback, ctx in callbacks:
            self._loop.call_soon(callback, self, context=ctx)
```

From previous section, we have seen the `Future` was scheduled into the event loop via `loop.ensure_future`. ***"If the argument is a Future, it is returned directly."*** So when the `Future` is scheduled in the event loop, there is almost no `callback` scheduled, until the `future.set_result` is called. (Almost no `callback` because there is a default `callback` `_run_until_complete_cb` added as we have seen previous section)

Task

Because `_PyFuture = Future`, `Task` is just a derived class of `Future`. The task of a `Task` is to wrap a `coroutine` in a `Future`. LINK

In the constructor, we see that the `Task` schedules a `callback` `self.__step` in the event loop. The `task.__step` is a long method, but we should just pay attention to the `try` block and the `else` block since these two are the ones mostly likely to be executed.

Here we see the `coroutine.send` method again. Each time we call `coroutine.send` in the `try` block, we get a `result`. In the `else` block, we always have another `self._loop.call_soon` call. We do this in a trampoline fashion until `Coroutine` runs out of results to `send`.

Asyncio HOWTO

Gather

`asyncio.gather(*coros_or_futures, return_exceptions=False)` automatically wraps each coroutine with a task and run a bunch of tasks concurrently. For example, let't look at how make multiple requests with `aiohttp`

```
import asyncio
import aiohttp
import time

def async_timed(func):
    async def wrapper(*args, **kwargs):
        t1 = time.perf_counter()
        res = await func(*args, **kwargs)
        print(f"Function {func.__name__} took {time.perf_counter
        return res
    return wrapper

@async_timed
async def fetch_status(session, url):
    async with session.get(url) as res:
        return res.status

@async_timed
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(10)]
        requests = [fetch_status(session, url) for url in urls]
        status_codes = await asyncio.gather(*requests)
        print(status_codes)
```

```
asyncio.run(main())

>>> Function fetch_status took 0.822050248999858
>>> Function fetch_status took 0.8216937839997627
>>> Function fetch_status took 0.8215144499999951
>>> Function fetch_status took 0.821447740000167
>>> Function fetch_status took 0.822015824999653
>>> Function fetch_status took 0.8217489159997058
>>> Function fetch_status took 0.8277271550000478
>>> Function fetch_status took 0.8247614050001175
>>> Function fetch_status took 0.824371204000272
>>> Function fetch_status took 0.825372482000148
>>> [200, 200, 200, 200, 200, 200, 200, 200, 200, 200]
>>> Function main took 0.8309274680000271
```

How to deal with exceptions in gather ?

By default if any exception happened in gather, it **will not** stop other tasks from execution. What is more, if parameter `return_exceptions=False` **(by default)** only first exception will be returned. And then we simply can iterate over results and filter them by type:

```
results = await asyncio.gather(*tasks, return_exceptions=True)
exceptions = [result for result in results if isinstance(result,
successful_results = [result for result in results if not isinst
```

There is one more lack of this of this approach. Let’s suppose we have many requests for ine server and if one request is failing it would be good to cancel all another, but it is difficult to do, because our coroutines are wrapped with tasks and running asynchronous.

What is more, we need to wait for all tasks to be finished to start working with results.

as_completed

`as_completed( fs , *, timeout =None)`

This function can take a list of coroutines and returns an iterator of future objects over which we can iterate and process results when they are ready (**need to apply await**).

For realistaion - LINK

```
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(10)]
        requests = [fetch_status(session, url) for url in urls]
        for finished_task in asyncio.as_completed(requests):
            print(await finished_task)
```

We also can add `timeout`, so to control overall time of execution and if it exceeds then raise `TimeoutError`

```
for finished_task in asyncio.as_completed(requests, timeout=2):
    try:
        print(await finished_task)
    except asyncio.TimeoutError:
        print('Timeout happened')
```

Main disadvantage is that the return order is not determined and if we need to rely on it we can’t.

The second disadvantage is that even when `timeout` happend all created tasks will run in background.

wait

`wait(fs, *, timeout=None, return_when=ALL_COMPLETED)`

This function is similar to gather, but it gives us more precise control. It returns 2 sets: `completed` (**success or exception**) and pending. Also, when timeout exceeds, no exception returns, just that tasks will be in pending set. Also, we can return results depending on `return_when`:

- ALL_COMPLETED
- FIRST_EXCEPTION

- FIRST_COMPLETED

> ***It accepts only futures or tasks***

```
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(10)]
        requests = [asyncio.create_task(fetch_status(session, u
        done, pending = await asyncio.wait(requests)
        for done_task in done:
            print(await done_task)
```

But what if there any exception encountered ?

```
### All code like upper
for done_task in done:
    if done_task.exception() is None:
        print(done_task.result())
    else:
        ### some logic to handle exception
```

Example with `FIRST_COMPLETED` parameter

```
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(10)]
        pending = [asyncio.create_task(fetch_status(session, ur
        while pending:
            done, pending = await asyncio.wait(requests, return_
            for done_task in done:
                print(await done_task)
```

wait_for

`wait_for(` `fut` `,` `timeout` `)` waits for the single Future or coroutine to complete, with timeout

If timeout happens, this coroutine will be canceled

```
async def foo():
    asyncio.sleep(1)

async def main():
    await asyncio.wait_for(foo(), timeout=2)

asyncio.run(main())
```

Sources

- https://masnun.com/2015/11/13/python-generators-coroutines-native-coroutines-and-async-await.html#comments
- https://github.com/timofurrer/awesome-asyncio
- Метью Фаулер “Asyncio та конкурентне програмування на Python”
- https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-1
- https://tenthousandmeters.com/blog/python-behind-the-scenes-12-how-asyncawait-works-in-python/
- https://ru.stackoverflow.com/questions/902586/asyncio-Отличие-tasks-от-future
- https://www.youtube.com/watch?v=1LTHbmed3D4 - This one is a series of videos
- https://leimao.github.io/blog/Python-AsyncIO-Event-Loop/
- https://leimao.github.io/blog/Python-AsyncIO-Awaitable-Coroutine-Future-Task/
- https://leimao.github.io/blog/Python-AsyncIO-Asynchronous-IO/