aiochan Documentation

Release 0.2.0

Ziyang Hu

Aug 25, 2018

Contents

Hello! We are excited to bring you what we consider the best concurrency control library available for Python.

You can start learning how to use aiochan now by following our tutorial, or, if you are already experienced in Golang or Clojure's core.async, we have a ten-minutes introduction for you that can help you get to speed immediately.

We hope you find that aiochan can boost your productivity greatly, as we do, and happy coding!

Note: you can try this tutorial in .

CHAPTER 1

A ten-minutes introduction

You will need to import the module `aiochan` and `asyncio` first:

```
In [2]: import aiochan as ac
        import asyncio
```

A channel is like a golang channel or a Clojure core.async chan. Creating a channel is simple:

```
In [3]: c = ac.Chan()
        c

Out[3]: Chan<_unk_0 140697829983528>
```

In the following examples, we use `ac.run` to run the main coroutine. You can also run asyncio loops directly.

We can call `await c.put(v)` to put value into the channel, `await c.get()` to get value from the channel, `c.close()` to close the channel, and `ac.go(...)` to spawn a coroutine inside another coroutine:

```
In [5]: async def producer(c):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                still_open = await c.put('product ' + str(i))
                if not still_open:
                    print('producer goes home')
                    break


        async def consumer(c):
            while True:
                product = await c.get()
                if product is not None:
                    print('obtained:', product)
                else:
                    print('consumer goes home')
                    break

        async def main():
            c = ac.Chan()
            ac.go(producer(c))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
consumer goes home
producer goes home
```

Channel works as an async iterator:

```
In [6]: async def producer(c):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                still_open = await c.put('product ' + str(i))
                if not still_open:
                    print('producer goes home')
                    break


        async def consumer(c):
            async for product in c:
                print('obtained:', product)
            print('consumer goes home')

        async def main():
            c = ac.Chan()
            ac.go(producer(c))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
consumer goes home
producer goes home
```

`select`, which is the whole point of channels, works as in golang or `alt!` in Clojure's core.async to complete one and only one operation non-deterministically:

```
In [8]: async def worker(out, stop, tag):
```

```
            i = 0
            while True:
                i += 1
                await asyncio.sleep(0.1)
                result, c = await ac.select(stop, (out, '%s-%s' % (tag, i)), priority=True)
                if c is stop:
                    print('%s stopped' % tag)
                    break

        async def consumer(c, stop):
            while True:
                result, c = await ac.select(stop, c, priority=True)
                if c is stop:
                    print('consumer stopped')
                    break
                else:
                    print('received', result)

        async def main():
            c = ac.Chan()
            stop = ac.Chan()
            for i in range(3):
                ac.go(worker(c, stop, 'worker%s' % i))
            ac.go(consumer(c, stop))
            await asyncio.sleep(0.6)
            stop.close()
            await asyncio.sleep(0.2)

        ac.run(main())
```

```
received worker0-1
received worker1-1
received worker2-1
received worker0-2
received worker1-2
received worker2-2
received worker0-3
received worker1-3
received worker2-3
received worker0-4
received worker1-4
received worker2-4
received worker0-5
received worker1-5
received worker2-5
consumer stopped
worker0 stopped
worker1 stopped
worker2 stopped
```

Channels can use some buffering to implement back-pressure:

```
In [10]: async def worker(c):
             i = 0
             while True:
                 i += 1
                 await asyncio.sleep(0.05)
                 print('producing', i)
                 await c.put(i)
```

```
async def consumer(c):
    while True:
        await asyncio.sleep(0.2)
        result = await c.get()
        print('consuming', result)

async def main():
    c = ac.Chan(3)
    ac.go(worker(c))
    ac.go(consumer(c))
    await asyncio.sleep(1)

ac.run(main())
```

```
producing 1
producing 2
producing 3
consuming 1
producing 4
producing 5
consuming 2
producing 6
consuming 3
producing 7
consuming 4
producing 8
```

Sliding and dropping buffers are also available.

That's all the basics, but there are much more: functional methods, combination patterns, pipelines, thread or process-based parallelism and so on. Read the in-depth tutorial or the API documentation to find out more.

CHAPTER 2

A beginner-friendly tutorial

In this tutorial we will learn how to use CSP with *aiochan*. We will start from the very basics.

Note: you can try this tutorial in .

2.1 What is concurrency

Aiochan is a python library for CSP-style concurrency in python. So what is concurrency? Why do we need concurrency? If the usual python programs that we are familiar with are "non-concurrent", aren't we doing just fine?

By far the most important justification for the need of concurreny is that *concurrency enables our programs to deal with multiple, potentially different, things all at once*. And no, we are not doing fine without concurrency. For example, suppose you are writing a webserver. Let's say that your code goes like this:

```
while True:
    parse request from client
    do something with the request (which takes time)
    return the result to the client
```

This is all very good. It works, but if this is the *outmost loop* of your server, then it is obvious that at most one client can be served at any one instant: clients effectively come in by a queue, and one can only be served when the previous one is done. If a client requires doing an operation that takes, say, ten minutes to complete, then the other clients will not be too happy.

"I have been writing python webservers using non-concurrent codes not too different from above for a long time, and it is definitely *not* true that only one client is served at any one instant". Well, most likely you are using some web frameworks and it is the framework that controls the outmost loop. In other words, your framework is managing all your concurrency whilst presenting a non-concurrent façade to you. By learning to write concurrent programs yourself, not only will you have the ability to fix your webserver when things go wrong (most often it is not bugs, but instead you are not honouring your framework's assumptions about what *your* code is supposed to be doing), but a great many opportunities of novel software architectures and applications also open up to you. The *potential* of building something greater than everything that comes before is what we have been always after, no?

So, concurrency means the ability to "deal with multiple things at once". Let's say that you have all the things you need to do written on a to-do list. And you are only a single person. Can you still have concurrency in completing your to-do list? Yes. Concurrency only means that you do not need to take the first item off the list, do it *and wait for the result*, then start with the second item. Your first item might well be "watch the news at 9am, then watch the news at 9pm, and find out what new things have happened since 9am". In this case non-concurrent behaviour means sitting there doing nothing after watching the 9am news until 9pm. As long as you *switch context* and starting doing something after the morning news you are in the realm of concurrent behaviour. You need neither a group of cronies to whom you can delegate your news watching, nor the rather unusual ability to watch two programs at once and perfectly understanding both, to concurrently perform tasks. Having a group of cronies doing stuff for you is a form of *parallelism*. Concurrency *enables* parallelism (it is useless to have many cronies if you need to wait for any one of them to complete their work before assigning work to the next one), but parallelism is not *necessary* for concurrency. Parallelism usually (but not always) makes things go faster. Concurrency can also make things go faster *even without parallelism*. In the case of computers, you *do not* need to have multiple processors to benefit from concurrency.

Now back to our webserver example. How do you write a concurrent outmost loop then? By analogy with the todo list example, you want somehow switch context and continue with the next client when you cannot fulfill the client's request immediately. It is a good exercise to try to implement it yourself before continuing.

Ok, back with your concurrent webserver implementation? Well, if you have not seen anyone else doing it before, you are probably still thinking in quite concrete terms like "requests", "accesses to databases", "file reading and writing" (if these are the things your webserver is doing), and trying to make these *concrete* tasks concurrent will feel like juggling ten balls at once – the solution is brittle, and you probably won't enjoy doing them everyday.

Here is the place for concurrent libraries, frameworks, or language constructs: they provide you with concurrent *abstractions*, and as long as you follow certain rules, they enable you to have the benefit of concurrency without the need of professional juggling training. *Aiochan* is such a library.

To recap:

- Concurrency enables you to deal with multiple things at once.
- Concurrency has the potential to decrease latency and increase throughput.
- Concurrency is not parallelism but enables it.
- Concurrency frameworks, libraries and language constructs provide abstractions that enable *you* to take advantange of concurrency without writing complicated and brittle code.

Note: you can try this tutorial in .

2.2 What is CSP

We have said that *aiochan* is a CSP-style concurrency library, and we explained what concurrency means. So what is CSP? CSP stands for *Communicating Sequential Processes*, and understanding what the three words in turn mean will get us a long way towards understanding how to use aiochan.

Let's begin with the last word: *processes*. Immediately, we have encountered an opportunity for great confusion: there are simply too many things in computing that are called "processes" at different times, by different people, and meaning subtly different things. Without dwelling on dictionary definitions, let's agree that in this section, a *process* just means a group of computer code that executes fairly independently from other codes and from the outside world, or, a group of code that you can mentally think of as a whole entity. The quintessential example is that of a function: a function as a process goes from taking in arguments from the caller, and ends when returning to the caller. A better word might be "task" here, but let's just stick with the original wording.

So for us, a *process* is something that is logically like a function, for now. What is a *sequential* process, then? If you read the word literally, it means that statements or expressions in your function are executed or evaluated in strict order, from top to bottom. Now this is also problematic: we have so-called control statements like `while`, `for`, etc., which are useful *because* they disrupt the sequential flow of your function. However, when your program is running, and all variables and arguments have concrete values, it is true that your statements in a function are executed one by one, in some *deterministic* order as specified by the control statements in the function. *Deterministic* is the keyword here: it is possible to *predict* what happens next by knowing the current state. In this section, we will use the word *sequential* in this sense: deterministically equivalent to sequential execution.

For example, let's look at the following snippet:

```
In [1]: x = 10
        x += 10
        x *= 2
        x -= 7
        print(x)
```

```
33
```

The above calculates `((10 + 10) * 2) - 7 = 33` and is sequential. If your programming language instead calculates `((10 * 2) + 10) - 7 = 3` then you have some serious issues. So sequential programs are good, it is what we as humans expect.

However, it is actually very easy to have non-sequential, or non-deterministic programs. Let's first refactor the above program:

```
In [2]: x = 10

        def f():
            global x
            x += 10
            x *= 2
            x -= 7
            return x

        print(f())
```

```
33
```

So far so good. But suppose you have two instances of the `f` process executing *concurrently*. In the following, we illustrate the interleaving of statements due to concurrency by putting two copies of the function side by side, and the execution order by the order that the statements appear:

```
x = 10

def f1():                        | def f2():
    global x # x == 10           |
                                 |     global x # x == 10
    x += 10 # x == 20            |
                                 |     x += 10 # x == 30
    x *= 2   # x == 40           |
                                 |     x *= 2   # x == 80
    x -= 7   # x == 73           |
    return x                     |
                                 |     x -= 7   # x == 66
                                 |     return x

print('f1 ->', f1())
print('f2 ->', f2())
```

We will get the results:

```
f1 -> 73
f2 -> 66
```

In this example, if you are only in control of `f1` you will be very much baffled. As you can try for yourself, by tweaking the order further you can get other results. This is despite the fact that within each function itself the sequence of statements is the same as before. So in our lingo, *within a single process*, the execution is now *non-deterministic*. We also call such processes *non-sequential* in order to align with the name CSP, although this terminology is obviously prone to misunderstanding.

In this case, the fix is actually not that hard: *don't modify global variables*. Any modifications you do must be local to your process. In functional languages, it is sometimes enforced that you cannot make any modifications at all — any computation you do just returns new values without stepping on the old values. However, we are writing python, and in python, such restriction is both unnecessary and unnatural. We only need to disallow operations that can interfere with other processes.

Now, you ask, what disturbed minds would write something like our `f`? Well, be assured that that people who wrote `f` habour no ill intensions. The reason that they reach for global variables is most often the need for input/output, or IO (note that the concept of IO is much broader than file or network accesses). We need to get stuff into our process to compute, and we need to notify other processes who are also computing what our results are.

Indeed, IO is the whole point of computation: *we*, at our keyboards (or touch screens, or whatever your newest VR/AR interaction devices), input something for the computer to compute, the the computer returns the results to *us*. Programs without IO are pretty useless. Using global variables for IO is also rather convenient: we just take something (input) from predetermined boxes (or memory addresses), and when we are done, just put the result into some other predetermined boxes. Other processes, by inspecting the boxes, will know what we have done. At the lowest level, this is actually what our current computer architecture dictates. A "pure" function that "returns" something without reference to an address *is* an illusion. But unfortunately, as we have seen, this crude arrangement results in processes stepping on each other and chaos if there are no rules.

The old and mainstream solution is that we put stickers on the boxes, or locks on memory addresses, when we want to operate on them: "in use — don't touch until I'm done!" This solves the problem, but using locks and similar *concurrency primitives* turn out to be rather delicate and error-prone. Suppose you and your friend both want to operate on two boxes A and B. You go forward and put your sticker on A, meanwhile your friend has already put his sticker on B. Now both of you are stuck: unless one of you back off, no one can go forward. Preventing such *deadlocks* means having a whole lot of disciplines and guidelines to follow — more training to become professional jugglers!

Is there a way out of this? Is there a way to avoid arduous juggler training while still doing concurrency? Yes, and this is what the *communicating* part of CSP says.

The basic idea is this: when doing computations that must involve IO, instead of boxes, we use *meeting points*, or *rendezvous*. For example, you and your friend both want a book. Instead of putting the book in a box so that both of you can do whatever you want with it whenever you want (and risking the book to be stolen), you just take the book away and do your processing with it. After you are satisfied, you and your friend *meet together* and you *hand off* the book to your friend. Once your friend has the book, she can do anything she wants with it, while you can no longer do anything with it at all. There is no longer any stepping over. If you *really* want your book again, you must arange with your friend for a hand-off again.

Such an arrangment is psychologically familiar (it is how private properties work), and it actually solves the majority of concurrency problems. These meeting points give us back *sequential process*: things look as if we are in a non-concurrent program, and the only surprises are where we expect them to be: at the meeting points, just like we expect to find something we don't already know when we read a file in. No stickers. No locks. Much less opportunities for deadlocks.

Not only does communicating over rendezvous solve the majority of problems that traditionally require the use of lock, it also solves these problems while *respecting the privacy, or abstraction barriers*, of the participating processes. Consider the box-book example again. If we want to use stickers to solve it, you and your friend both have to agree on a strategy, for example, always start with box A. Now you are both opening yourselves up to each other, letting the other know things about how *you* operate, which you may be reluctant to with someone you just met. By contrast, when using rendezvous for communication, the existence of rendezvous and whether you intend to use it for reading or writing is often sufficient for correct execution. The abstraction barrier is respected!

The rest of this tutorial will go into much more details in how to go about setting up and honouring rendezvous, which in the context of *aiochan*, is called a *channel*, or `Chan`. But first, we need to deal with some environment setups in the next section.

To recap, in the context of CSP (*Communicating Sequantial Processes*):

- Processes are group of codes that can be considered as an independent entity.
- Sequential processes are processes that operate in deterministic order producing deterministic results, without danger of stepping over each other.
- Communicating sequantial processes are sequantial processes that do their IO by rendezvous only.
- CSP style concurrency enables natural program logic resembling non-concurrent codes, respects abstraction barriers, while at the same time eliminating most of the dangers of deadlocks.

Note: you can try this tutorial in .

2.3 Coroutines and event loops

Programming languages begin in different ways, with different goals. Unfortunately, in the case of python, concurrency isn't one of the goals (unlike, say, in the case of Erlang and Elixir). As a result, concurrency in python feel a bit foreign, cumbersome and unnatural.

In particular, when you start your python interactive interpreter, or when you run your python script, your program begins in an environment that can be roughly described as a single-threaded process, with no obvious ladder that allows you to climb up onto a concurrent environment. So what are "processes" and "threads"?

"Process" here means something different from the "process" in CSP: here it refers to an instance of your program that is executed for you by the operating system. A process has its own memory space and file descriptors provided by the operating system, and these are by default isolated from the other processes on the same system.

A process may have one or more *threads of execution* running at the same time, with each thread executing its part of the code in sequence, but different threads can share memory. If the sharing is not done carefully and in a principled way, as we have seen it will lead to non-deterministic results.

Python supports threads by providing the `thread` and `threading` module. But these are not too useful (compared to other languages, at least): in python, the danger of threads stepping on each other is so great that the default implementation, CPython, has a global interpreter lock, or GIL, that ensures that only a single python statement can execute at a given instance! Even with such a big fat lock always in place, we still need locks in order to prevent unwanted interleaving! So the GIL prevents us from using multiple processors. Locking has overheads. To make things still worse, python schedules thread execution in a somewhat unintuitive manner which results in favouring slow (or CPU-intensive) operations over fast ones, the opposite of what most operating system does. The end result: python code utilizing threads often runs *slower* than those that do not.

Python also supports spawning processes within python itself using the `multiprocessing` module. But by default processes don't share memory or resources, hence inter-process communicating is restricted and cumbersome. The overhead of using processes is even greater than using threads. Well, the picture isn't very pretty.

Still, not being able to deal with multiple things at once is stupid. Considering the situation we are in, it seems the best way to go forward is to have something that is single-threaded (lower overhead, hopefully) that can imitate multiple threads of execution. And there is something along this way built into the language since python 3.4, it is called `asyncio`.

Compared to plain python, `asyncio` utilizes two further keywords (since python 3.5, at least): `async` and `await`. `async` is applied to functions (and methods). An example:

```
In [2]: async def silly(x):
            print('executing silly with', x)
            return x+1
```

It seems normal enough. But when you call it:

```
In [3]: silly(2)
```

```
Out[3]: <coroutine object silly at 0x7f902c9c0b48>
```

It seems that the function doesn't execute but instead returns something called a coroutine. Calling async function is a two step process: first you call it normally and obtain a coroutine, and the coroutine needs to be given to some scheduler, or *event loop*, for execution. The function `aiochan.run` will do the scheduling and executing part for you:

```
In [4]: import aiochan as ac
        ac.run(silly(2))
```

```
executing silly with 2
```

```
Out[4]: 3
```

Every call to `ac.run` creates a new event loop, which runs until the passed in async function finishes executing.

All this ceremony of using `async` and running coroutines in a strange way sets up the stage for using the `await` keyword:

```
In [5]: import asyncio

        async def count(tag, n_max=3):
            i = 0
            while i < n_max:
                await asyncio.sleep(0.5)
                i += 1
                print(tag, i)

        ac.run(count('counter:'))
```

```
counter: 1
counter: 2
counter: 3
```

Whatever after the `await` keyword must be an *awaitable*, which roughly says that "this computation here will eventually produce something, but maybe not right now, and while waiting you may go off and do something else: the scheduler will let you continue when the result is ready".

Let's see what happens when we run two counters (remember that the function `count`, when called, produces a coroutine, which is an awaitable):

```
In [6]: async def main():
            await count('counter a:')
            await count('counter b:')

        ac.run(main())
```

```
counter a: 1
counter a: 2
counter a: 3
counter b: 1
counter b: 2
counter b: 3
```

Hmm . . . this doesn't look very concurrent: the second counter starts counting only after the first counter finishes. But this is what we asked for: we awaited for the completion of the first counter!

To make the two counters execute together, we use the `aiochan.go` function, which takes a coroutine and schedules it for execution but do not wait for the result:

```
In [7]: async def main():
            ac.go(count('counter a:'))
            await count('counter b:')

        ac.run(main())
```

```
counter b: 1
counter a: 1
counter b: 2
counter a: 2
counter b: 3
counter a: 3
```

Much better now. Note that you *must* pass the coroutine to `aiochan.go` for execution: calling the function itself has no effect (other than a possible warning):

```
In [8]: async def main():
            count('counter a:')
            await count('counter b:')

        ac.run(main())
```

```
/home/zh217/.pyenv/versions/3.6.6/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning


counter b: 1
counter b: 2
counter b: 3
```

What happens we replace both counter calls with `aiochan.go`?

```
In [9]: async def main():
            ac.go(count('counter a:'))
            ac.go(count('counter b:'))

        ac.run(main())
```

Nothing happens! Remember that `ac.run` returns when the coroutine passed in returns, and our `main` returns after having two counters scheduled for execution, without actually executing them!

To make this clearer, note that if we sleep in the `main` function at the end, the two counters will be executed:

```
In [10]: async def main():
             ac.go(count('counter a:'))
             ac.go(count('counter b:'))
             await asyncio.sleep(3)

         ac.run(main())
```

```
counter a: 1
counter b: 1
counter a: 2
counter b: 2
counter a: 3
counter b: 3
```

If you have done thread-based programming before, you may think now that asyncio is no different from threading. This is not true. To illustrate, consider:

```
In [11]: async def main():
             async def work():
                 print('do work')
             print('before')
             ac.go(work())
```

```
            print('after')

        ac.run(main())
```

```
before
after
do work
```

What you get is *always* `before`, `after`, and `do work`, in that order. In some languages, using thread, it is possible to get garbled texts, since the various calls to `print` (or whatever it is called) can step on each other. By contrast, asyncio event loops uses only a single thread, and it is guaranteed that unless you `await`, nothing else will get in your way when you are executing. (If you read the documentations for asyncio, you will find that even things like locks and semaphores are marked "not thread-safe" — they are only safe with respect to the non-interrupting guaruantees provided by asyncio.)

So asyncio guarantees "no break unless await", in other words, it implements *co-operative multitasking*, in constrast to the *pre-emptive multitasking* provided by threads (and your operating system) where your work may be interrupted at any time. Programming co-operative multitasking is in general easier, however it is sometimes necessary to explicitly give up control to the scheduler in order for other tasks to have a chance to run. In aiochan, you can await for `aiochan.nop()` to achieve this:

```
In [12]: async def main():
             async def work():
                 print('do work')
             print('before')
             ac.go(work())
             await ac.nop()
             print('after')

         ac.run(main())
```

```
before
do work
after
```

Note the order. Also note that in this case, theoretically the order *isn't* guaranteed — that you always get this order back should be considered an implementation detail.

Now you know how to make coroutines and run them. That roughly corresponds to the "sequential processes" that we talked about before (and remember not to touch global states). In the next section, we will learn about the "communicating" part.

To recap:

- Python was not designed for concurrency.
- There are a number of ways to do concurrency in python: processes, threads, and asyncio event-loops.
- Asyncio event loops are single-threaded schedulers responsible for executing coroutines.
- Coroutine functions are made with `async` and `await` keywords. No interleaving of execution can occur unless an `await` keyword is encountered.

Useful functions:

- `aiochan.run`
- `aiochan.go`
- `aiochan.nop`
- `asyncio.sleep`

There is also `aiochan.run_in_thread`, which is recommended for scripts. `aiochan.run` is recommended when programming interactively.

2.3.1 Note about `ac.run` in Jupyter notebooks

If you use `ac.run` in Jupyter notebooks to run examples, sometimes you will see warnings saying something like:

```
Task was destroyed but it is pending!
```

In the notebook, this is mostly harmless and it is due to our not always closing all our coroutines when our main coroutine exits. We do this in order to make our examples simple, and to avoid using constructs not already introduced. However, *in production*, any warning is a cause for concern.

2.3.2 Appendix: running async functions without aiochan

In our exposition we used the function `aiochan.run` to run all our async functions. How to do it with native python libraries? We use asyncio event loops to do the execution:

```
import asyncio

loop = asyncio.get_event_loop()
result = loop.run_until_complete(silly(2))
print('the result is', result)
```

You can try running the above code. What you get depends on how you run it: if you run it in a script or in an interactive interpreter, then you will see printed:

```
executing silly with 2
result: 3
```

However, if you run it in jupyter notebooks or jupyterlab, there is a possibility that you will get an exception thrown at your face (or maybe not, it all depends):

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-6-69c4a90e423b> in <module>()
      2 
      3 loop = asyncio.get_event_loop()
----> 4 result = loop.run_until_complete(silly(2))
      5 print('the result is', result)

~/.pyenv/versions/3.6.6/lib/python3.6/asyncio/base_events.py in run_until_
↪complete(self, future)
    453         future.add_done_callback(_run_until_complete_cb)
    454         try:
--> 455             self.run_forever()
    456         except:
    457             if new_task and future.done() and not future.cancelled():

~/.pyenv/versions/3.6.6/lib/python3.6/asyncio/base_events.py in run_forever(self)
    407         self._check_closed()
    408         if self.is_running():
--> 409             raise RuntimeError('This event loop is already running')
    410         if events._get_running_loop() is not None:
    411             raise RuntimeError(
```

(continues on next page)

(continued from previous page)

```
RuntimeError: This event loop is already running
```

So we already have a loop running? Ok, it is still possible to proceed in this case

```
In [13]: import asyncio

         loop = asyncio.get_event_loop()
         result = loop.create_task(silly(2))
         print('the result is', result)
```

```
the result is <Task pending coro=<silly() running at <ipython-input-2-709c439f84a4>:1>>
executing silly with 2
```

So apparently our async function is executed now, but now we only get a task back, not the result itself! To get the result:

```
In [14]: result.result()
```

```
Out[14]: 3
```

. . . which seems to be fine, but that is only because you are executing it interactively. If you put this line directly below `print`, you most certainly will get:

```
---------------------------------------------------------------------------
InvalidStateError                         Traceback (most recent call last)
<ipython-input-17-cda1a6adb807> in <module>()
      4 result = loop.create_task(silly(2))
      5 print('the result is', result)
----> 6 result.result()

InvalidStateError: Result is not set.
```

which tells you that you are calling the function too soon! You will need to wait a little bit (but if you do it wrong it will deadlock), or you create some future and set the result use a callback. If you really want to figure it out you can read the python documentations.

So now you believe me when I say that doing concurrency in python feels foreign, cumbersome and unnatural. Running everything in a script is a solution, but one of the appeal of python is its interactivity.

You can also replace calls to `aiochan.go` with `asyncio.get_running_loop().create_task`, but . . . what a mouthful! `asyncio.ensure_future` is also a possibility, but in addition to its questionable name, its use in spawning tasks for execution is deprecated in python 3.7 in favour of `asyncio.create_task`. However, `asyncio.create_task` doesn't exist prior to python 3.7. So . . . if you intend to use `aiochan` at all, we urge you to stay with `aiochan.go`.

Note: you can try this tutorial in .

2.4 Channels

Now we know how to make corountines and schedule them for execution. As we said before, for the coroutines to do IO safely and in a principled way, we will use meeting points, which in `aiochan` is called `Chan`, for "channel". Constructing a channel is easy:

```
In [2]: import aiochan as ac
        import asyncio
        c = ac.Chan()
        c
```

```
Out[2]: Chan<_unk_0 140664273474752>
```

Now suppose we have a producer that can be tasked to producing items, and a consumer that can consume items. The IO in this case is the outputs of the producer outputs, and the inputs of the consumer, and these two are linked in a channel. In code:

```
In [3]: async def producer(c):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                await c.put('product ' + str(i))

        async def consumer(c):
            while True:
                product = await c.get()
                print('obtained:', product)

        async def main():
            c = ac.Chan()
            ac.go(producer(c))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
```

We see that `Chan` has two methods `put` and `get`. `put` is used to put stuff into the channel, and `get` is for getting stuff out. Both of these return awaitables, signaling that doing IO with channels involves potential waiting, as two parties need to come together for either of them to proceed. Awaiting a `get` produces the value that is just `put` into the channel.

In `aiochan`, you cannot `put` something turns out to be `None` into a channel (other falsy values such as `0`, `0.0`, `[]`, `{}`, `False` are ok). The reason is that a channel can be closed, and we need to signal somehow to the users of channel that it is closed, and we use `None` for the signal. Another possibility is throwing exceptions, but throwing exceptions in async code quickly gets *very* confusing. So, following Clojure's core.async, we don't allow `None` values in channels.

Speaking of closing channels, note that in our previous example, `main` just walks away when it is determined that everyone should go home. But `producer` and `consumer` are just left there dangling, which is very rude of `main`. Closing the channel is a polite way of notifying them both:

```
In [4]: async def producer(c):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                still_open = await c.put('product ' + str(i))
                if not still_open:
                    print('producer goes home')
                    break
```

```
        async def consumer(c):
            while True:
                product = await c.get()
                if product is not None:
                    print('obtained:', product)
                else:
                    print('consumer goes home')
                    break

        async def main():
            c = ac.Chan()
            ac.go(producer(c))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
consumer goes home
producer goes home
```

We see that after the channel is closed with `c.close()`, awaiting a `get` will produce a `None`, whereas awaiting a `put` will produce `False` (before closing it will return `True`).

By the way, on python 3.6 and above, we can simplify our consumer a bit: here we are just iterating over the values in the channel one by one, which is exactly what an asynchronous iterator does. So we can write

```
In [5]: async def producer(c):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                still_open = await c.put('product ' + str(i))
                if not still_open:
                    print('producer goes home')
                    break


        async def consumer(c):
            async for product in c:
                print('obtained:', product)
            print('consumer goes home')

        async def main():
            c = ac.Chan()
            ac.go(producer(c))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
consumer goes home
producer goes home
```

It is also no longer necessary to test whether `product` is None: the iteration stops automatically when the channel is closed.

Note that in `aiochan`, a channel is just an object — in some circles, this is called a "first-class construct". This means that it can be passed as arguments to functions (which we just did), returned from functions, or stored in a datastructure for later use (unlike, say, in Erlang). It is even possible to go meta: a channel containing channels.

For example, we can make our producer producing the channel instead:

```
In [6]: async def producer():
            c = ac.Chan()

            async def work():
                i = 0
                while True:
                    await asyncio.sleep(0.1) # producing stuff takes time
                    i += 1
                    still_open = await c.put('product ' + str(i))
                    if not still_open:
                        print('producer goes home')
                        break

            ac.go(work())
            return c


        async def consumer(c):
            async for product in c:
                print('obtained:', product)
            print('consumer goes home')

        async def main():
            c = await producer()
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1
obtained: product 2
obtained: product 3
obtained: product 4
obtained: product 5
It is late, let us call it a day.
consumer goes home
```

```
producer goes home
```

But in this case, *not* letting the producer producing its own channel actually has benefit: we can easily have several producers working in parallel:

```
In [8]: async def producer(c, tag):
            i = 0
            while True:
                await asyncio.sleep(0.1) # producing stuff takes time
                i += 1
                still_open = await c.put('product %s from %s' % (i, tag))
                if not still_open:
                    print('producer %s goes home' % tag)
                    break


        async def consumer(c):
            async for product in c:
                print('obtained:', product)
            print('consumer goes home')

        async def main():
            c = ac.Chan()
            for i in range(3):
                ac.go(producer(c, 'p%s' % i))
            ac.go(consumer(c))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
obtained: product 1 from p0
obtained: product 1 from p1
obtained: product 1 from p2
obtained: product 2 from p0
obtained: product 2 from p1
obtained: product 2 from p2
obtained: product 3 from p0
obtained: product 3 from p1
obtained: product 3 from p2
obtained: product 4 from p0
obtained: product 4 from p1
obtained: product 4 from p2
obtained: product 5 from p0
obtained: product 5 from p1
obtained: product 5 from p2
It is late, let us call it a day.
consumer goes home
producer p0 goes home
producer p1 goes home
producer p2 goes home
```

This is call *fan-in*: different producers fanning their products into the same channel. We can also have *fan-out*:

```
In [10]: async def producer(c, tag):
             i = 0
             while True:
                 await asyncio.sleep(0.1) # producing stuff takes time
                 i += 1
```

```
                still_open = await c.put('product %s from %s' % (i, tag))
                if not still_open:
                    print('producer %s goes home' % tag)
                    break


        async def consumer(c, tag):
            async for product in c:
                print('%s obtained: %s' % (tag, product))
            print('consumer %s goes home' % tag)

        async def main():
            c = ac.Chan()
            for i in range(3):
                ac.go(producer(c, 'p%s' % i))
            for i in range(3):
                ac.go(consumer(c, 'c%s' % i))
            await asyncio.sleep(0.6)
            print('It is late, let us call it a day.')
            c.close()
            await asyncio.sleep(0.2) # necessary to wait for producer

        ac.run(main())
```

```
c0 obtained: product 1 from p0
c1 obtained: product 1 from p1
c2 obtained: product 1 from p2
c0 obtained: product 2 from p0
c1 obtained: product 2 from p1
c2 obtained: product 2 from p2
c0 obtained: product 3 from p0
c1 obtained: product 3 from p1
c2 obtained: product 3 from p2
c0 obtained: product 4 from p0
c1 obtained: product 4 from p1
c2 obtained: product 4 from p2
c0 obtained: product 5 from p0
c1 obtained: product 5 from p1
c2 obtained: product 5 from p2
It is late, let us call it a day.
consumer c0 goes home
consumer c1 goes home
consumer c2 goes home
producer p0 goes home
producer p1 goes home
producer p2 goes home
```

We see that works are divided between producers and consumers evenly automatically. Even if producers produce things at different rate, this fan-in, fan-out pattern will automatically do the right thing:

```
In [11]: async def producer(c, tag, interval):
             i = 0
             while True:
                 await asyncio.sleep(interval) # producing stuff takes time
                 i += 1
                 still_open = await c.put('product %s from %s' % (i, tag))
                 if not still_open:
                     print('producer %s goes home' % tag)
                     break
```

```
async def consumer(c, tag):
    async for product in c:
        print('%s obtained: %s' % (tag, product))
    print('consumer %s goes home' % tag)

async def main():
    c = ac.Chan()
    for i in range(3):
        ac.go(producer(c, ('p%s' % i), interval=(i+1)*0.1))
    for i in range(3):
        ac.go(consumer(c, 'c%s' % i))
    await asyncio.sleep(1)
    print('It is late, let us call it a day.')
    c.close()
    await asyncio.sleep(0.2) # necessary to wait for producer

ac.run(main())
```

```
c0 obtained: product 1 from p0
c1 obtained: product 1 from p1
c2 obtained: product 2 from p0
c0 obtained: product 1 from p2
c1 obtained: product 3 from p0
c2 obtained: product 2 from p1
c0 obtained: product 4 from p0
c1 obtained: product 5 from p0
c2 obtained: product 2 from p2
c0 obtained: product 3 from p1
c1 obtained: product 6 from p0
c2 obtained: product 7 from p0
c0 obtained: product 4 from p1
c1 obtained: product 8 from p0
c2 obtained: product 3 from p2
c0 obtained: product 9 from p0
It is late, let us call it a day.
consumer c1 goes home
consumer c2 goes home
consumer c0 goes home
producer p1 goes home
producer p0 goes home
```

We see that jobs are still divided evenly between consumers, but more jobs come from faster producers.

To recap:

- The construct for inter-coroutine communication is the channel.
- Getting and putting to channels facilitates IO between coroutines.
- Channels are first-class construct: we can pass them around, return them, or store them.
- Channels can be closed.
- `None` values are not allowed on channels.
- Strategically closing channels can be used for execution control.
- Fan-in and fan-out can be used for distributing works among different coroutines.

Useful constructs:

- `aiochan.Chan`

- `aiochan.Chan.put`
- `aiochan.Chan.get`
- `aiochan.Chan.close`

Note: you can try this tutorial in .

2.5 Select: the quitessential operation

Channels with their put and get operations can already be used to build rather complicated systems. Now we introduce the operation `select`, which hugely increases the expressive power of channels further.

Basically, if we have channels `c1`, `c2` and `c3` and we write

```
result = await select(c1, c2, c3)
```

then `result` will hold the result of one and only one `get` operation on `c1`, `c2` and `c3`. *Only one operation will be attempted*. If we have several operations that can be completed at the same time, only one will complete, and the non-completing ones *will not run at all*. This is in constrast with, say, `asyncio.wait`.

Let's have some examples:

```
In [2]: import asyncio
        import aiochan as ac

        async def main():
            c1 = ac.Chan(name='c1').add(1, 2, 3).close()
            c2 = ac.Chan(name='c2').add('a', 'b', 'c').close()
            c3 = ac.Chan(name='c3').add('x', 'y', 'z').close()

            result, chan = await ac.select(c1, c2, c3)
            print('the result is', result)
            print('the result is from', chan)

            async for v in c1:
                print('c1 still has value:', v)

            async for v in c2:
                print('c2 still has value:', v)

            async for v in c3:
                print('c3 still has value:', v)

        ac.run(main())
```

```
the result is 1
the result is from Chan<c1 140594564470264>
c1 still has value: 2
c1 still has value: 3
c2 still has value: a
c2 still has value: b
c2 still has value: c
c3 still has value: x
c3 still has value: y
c3 still has value: z
```

Here we have also used some new operations on channels:

- We can give names to channels: `Chan(name='some name')`,

- `ch.add(...)` adds elements to channels on the background when it is possible to do so,
- `close` closes the channel immediately, but all pending puts (here those by `add`) will still have an opportunity to complete,
- `add` and `close` can be chained as both these methods return the channel.

And for our `select`:

- it returns a tuple: the value together with the channel that is involved,
- if several operations can all be completed, which one is completed is non-deterministic (try running the above script several times to see).

Actually, it is not only get operations that can be `select`ed:

```
In [3]: async def receive(c):
            r = await c.get()
            print('received', r, 'on', c)

        async def main():
            c1 = ac.Chan(name='c1')
            c2 = ac.Chan(name='c2')

            ac.go(receive(c1))
            ac.go(receive(c2))

            await ac.nop()

            result, chan = await ac.select((c1, 'A'), (c2, 'B'))
            print('select completes on', chan)

        ac.run(main())
```

```
select completes on Chan<c2 140594564470264>
received B on Chan<c2 140594564470264>
```

we see that if we give an argument like `(chan, value)` it is interpreted as a put operation akin to `chan.put(value)`. Again, one and only one operation will complete. You can also mix get operations with put operations.

Also, if you are careful, you will have noticed that we have inserted a `nop` above. If it is not there, the `select` will always complete on `c1`. You may want to think about why.

The more non-trivial the application is, the more use of `select` you can find. One of its simplest use is for stopping many workers at once:

```
In [5]: async def worker(out, stop, tag):
            i = 0
            while True:
                i += 1
                await asyncio.sleep(0.1)
                result, c = await ac.select(stop, (out, '%s-%s' % (tag, i)), priority=True)
                if c is stop:
                    print('%s stopped' % tag)
                    break

        async def consumer(c, stop):
            while True:
                result, c = await ac.select(stop, c, priority=True)
                if c is stop:
                    print('consumer stopped')
                    break
                else:
                    print('received', result)

        async def main():
            c = ac.Chan()
            stop = ac.Chan()
            for i in range(3):
                ac.go(worker(c, stop, 'worker%s' % i))
            ac.go(consumer(c, stop))
            await asyncio.sleep(0.6)
            stop.close()
            await asyncio.sleep(0.2)

        ac.run(main())
```

```
received worker0-1
received worker1-1
received worker2-1
received worker0-2
received worker1-2
received worker2-2
received worker0-3
received worker1-3
received worker2-3
received worker0-4
received worker1-4
received worker2-4
received worker0-5
received worker1-5
received worker2-5
consumer stopped
worker0 stopped
worker1 stopped
worker2 stopped
```

Here stopping can actually be signaled by simply closing the fan-in-fan-out channel, but in more complicated situations (for example, closing down in response to *any one* of several conditions) `select` is essential.

We have also seen that `select` takes an argument `priority`, which defaults to `False`. Here we set it to true, so when several operations become completable at the same time, it is guaranteed that the leftmost one will complete. Here we use this priority `select` to make sure that the operation stops at the earliest instance.

There is also a `default` argument to `select`, which if set, will produce the set value immediately when none of the operations can be completed immediately, with `None` in the place where you usually find the completed channel. The following snippet completes the put only if it can be done immediately:

```
In [6]: async def main():
            ch = ac.Chan()
            result, c = await ac.select((ch, 'value'), default='giveup')
            if c is None:
                print(result)
                print('put cannot complete immediately and was given up')

        ac.run(main())
```

```
giveup
put cannot complete immediately and was given up
```

By now you should know how to use `select`. It certainly seems a simple enough operation to understand. However, `select` is non-trivial. What we mean by that is that, using only channels and put and get operations on channels, it is not possible to write a `select` clone that has the correct semantics. The semantics of `select` has three requirements:

- at least one operation is completed;
- at most one operation is completed;
- an operation is completed at the earliest possible time (no unnecessary waiting).

Writing an operation satisfying any two of the above is easy. But to satisfy all three, you need to submit your operations to the involved channels at the time of calling, and at the time of completion of any operation, you will need to notify all other operations to cancel themselves. Thus the semantics of `select` must be implemented inside `Chan`, not outside.

`select` is actually the whole point of `aiochan`: `asyncio` do provide us with futures, locks and things, which are somewhat like our channels superficially. But `select` is conspicuously missing. Channels are made to make `select` possible. Rob Pike, the inventor of golang, mentions `select` as the reason why channels in golang is provided by the language itself instead of as a library.

Another way of putting this is: in the hierarchy of concurrency operations, `select` is on the highest level of abstraction. Consider the following:

- unlike python, Java was designed with concurrency (with threads) in mind, so thread primitives exist from the beginning;
- but as working with the primitives were too low-level, `java.util.concurrent` was added as a libray;
- Clojure runs on the JVM so can use all the Java concurrency libraries. Clojure also adds its own flavour of concurrency-friendly constructs in the form of refs (atoms, agents, and even STM)
- BUT Clojure still needs `core.async` as a library, since writing a `select` that works well on all the previous stuff is not possible! (By the way, `select` is called `alt!`, `alts!`, `alt!!` and `alts!!` in core.async. Yes there are four of them.)

By the way, python has a built-in library called `select`, and a higher-level one doing essentially the same thing called `selectors`. But these libraries only work with files or sockets, not plain python objects, and the availability of the various operations in theses libraries depend on the operating system. That is because the library just offloads it work to system calls. Usually we think of system calls as pretty low level. How many times have you encountered some abstraction that is provided by the lower-level operating system but not by the higher-level programming language?

To recap:

- The `select` operator completes exactly one operation from the given operations,
- `select` can be used as a control structure,
- `select` is non-trivial.

Useful constructs:

- `select`
- `aiochan.Chan.add`
- Channel operations can be chained (more to come)

Note: you can try this tutorial in .

2.6 Channel buffering

Channels are used as meating points: a pair of put/get operations can only be completed when *both* involved parties are present at the same time. However, in practice, it is sometimes necessary to relax this requirement a little bit so that puts can complete immediately even when no one is there to get, and if previous puts are available, a get can complete without a put sitting there waiting. This behaviour where we further decouple put and get operations in time is called buffering.

In principle, buffering can be done without any further support from the library: we can have a pair of channels, `ch_in` and `ch_out`, acting as one, and a coroutine busy working in the background, promptly getting values from `ch_in` whenever they come in and store them onto some data structure, and at the same time feeding values to getters of `ch_out` whenever they come by. You can do this as an exercise (hint: use `select`).

However, to reduce clutter, and to improve performance, `Chan` has built-in support for buffer. In `aiochan`, buffering is always bounded: you have to decide at the onset how much pending stuff stored in your buffer you can tolerate. Some languages like Erlang nominally support unbounded buffering as the default, but the limit imposed by the operating system is always there.

Let's have an example:

```
In [2]: import asyncio
        import aiochan as ac

        async def main():
            c = ac.Chan(1)

            await c.put('a')
            result = await c.get()
            print('result', result)

        ac.run(main())
```

```
result a
```

As we can see, a buffered channel is created by having a positive number as the argument to the channel constructor. In this example, if there were no buffer, the example would deadlock: the first `await` would never complete.

The positive number to the constructor signifies the size of the buffer. In the example, the size is one, so if we have two puts in a row the example would block.

This is an example of *fixed length buffers*. The constructor call `Chan(1)` is actually a shorthand for `Chan('f', 1)`, `'f'` for fixed length. These buffers block on put when they are full.

Fixed length buffers are often used to implement back-pressure:

```
In [3]: async def worker(c):
            i = 0
            while True:
                i += 1
                await asyncio.sleep(0.05)
                print('producing', i)
                await c.put(i)

        async def consumer(c):
            while True:
                await asyncio.sleep(0.2)
                result = await c.get()
                print('consuming', result)

        async def main():
            c = ac.Chan(3)
            ac.go(worker(c))
            ac.go(consumer(c))
            await asyncio.sleep(1)

        ac.run(main())
```

```
producing 1
producing 2
producing 3
```

```
consuming 1
producing 4
producing 5
consuming 2
producing 6
consuming 3
producing 7
consuming 4
producing 8
```

Here, producers and consumers are working at different rates. We want to ensure the consumer always have something to work with, so producers have to work ahead of consumers, but we also want to ensure that producers don't work so fast that the consumers can never catch up. A buffer solves the problem well. Our buffering solution still works even if the time taken to produce/consume items are somewhat random: within bounds, appropriate buffering can ensure minimal waiting while preventing producing and consuming rates from diverging.

In situations that getters just can't keep up with putters *and* you definitely cannot tolerate blocking for producers (maybe because you don't control the producers), you have to make some compromise and use some other kinds of buffers which will *discard* some elements in exchange for non-blocking puts. We have built-in support for two of them: *dropping buffers* will just silent drop any more incoming puts when they become full:

```
In [4]: async def main():
            c = ac.Chan('d', 2) # 'd' for 'dropping'
            await c.put(1)
            await c.put(2)
            await c.put(3)
            c.close()
            async for v in c:
                print(v)

        ac.run(main())
```

```
1
2
```

Look: the last value is missing. We also have *sliding buffers*, which when full, will drop the *earliest* pending value:

```
In [5]: async def main():
            c = ac.Chan('s', 2) # 'd' for 'dropping'
            await c.put(1)
            await c.put(2)
            await c.put(3)
            c.close()
            async for v in c:
                print(v)

        ac.run(main())
```

```
2
3
```

At the beginning we have said that channels are used to circumvent the use of locks and semaphores so that our programs are easier to develop and easier to reason about. Well, sometimes locks and semaphores are the most natural solutions to a problem. And in such situations, *buffered channels can be used as locks and semaphores*.

An example:

```
In [6]: async def worker(lock, tag):
            while True:
                await lock.get()
                print('%s is now working' % tag)
                await asyncio.sleep(0.1)
                await lock.put(True)

        async def main():
            lock = ac.Chan(1).add(True)
            for i in range(10):
                ac.go(worker(lock, i))
            await asyncio.sleep(1)

        ac.run(main())
```

```
0 is now working
1 is now working
2 is now working
3 is now working
4 is now working
5 is now working
6 is now working
7 is now working
8 is now working
9 is now working
```

In 1 second, only 10 operations complete even though we have 10 workers, whose maximum productivity is 100 operations in 1 second. In the presence of the lock, work becomes serial.

Using a buffer size greater than 1 gives you a semaphore, which in our case increases the throughput:

```
In [7]: async def worker(lock, tag):
            while True:
                await lock.get()
                print('%s is now working' % tag)
                await asyncio.sleep(0.1)
                await lock.put(True)

        async def main():
            lock = ac.Chan(2).add(True, True)
            for i in range(10):
                ac.go(worker(lock, i))
            await asyncio.sleep(1)

        ac.run(main())
```

```
0 is now working
1 is now working
2 is now working
3 is now working
4 is now working
5 is now working
6 is now working
7 is now working
8 is now working
9 is now working
0 is now working
1 is now working
2 is now working
3 is now working
4 is now working
5 is now working
6 is now working
7 is now working
8 is now working
9 is now working
```

But why would you want to use channels *as* locks when you can use the builtin locks from `asyncio`? Consistency and flexibility. Remember `select`? Now we can `select` on locks! You can do all kinds of funky stuff with `select` and locks:

```
In [9]: import random

        async def worker(locks, tag):
            while True:
                _, lock = await ac.select(*locks)
                print('%s working' % tag)
                await asyncio.sleep(0.1)
                await lock.put(True)

        async def main():
            locks = [ac.Chan(1, name='lock%s' % i).add(True) for i in range(3)]
            for i in range(3):
                ac.go(worker(locks, 'worker-%s' % i))
            await asyncio.sleep(0.5)

        ac.run(main())
```

```
worker-0 working
worker-1 working
worker-2 working
worker-0 working
worker-1 working
worker-2 working
worker-0 working
worker-1 working
worker-2 working
worker-0 working
worker-1 working
worker-2 working
worker-0 working
worker-1 working
worker-2 working
```

Now the worker proceeds whenever it can get its hand on *any* of a sequence of locks. With 3 locks we got 15 units of work done in half a second. You can change to 2 locks, in which case only 10 units of work would be done.

To recap:

- Channels support buffering.
- Fixed length buffering blocks on put when full, whereas dropping and sliding buffering never blocks but may throw away items when full.
- Buffering can be used to implement back-pressure.
- Buffered channels can be used as locks and semaphores, and you can `select` on them.

Congratulations! Now you know *almost* everything you need to write non-trivial concurrency applications with `aiochan`. You are only limited by your imagination! Still, there are various *patterns* of concurrency programs that occur so often so that we have implemented them as additional functions and methods that you can readily use. None of them is essential, but using the provided *convenience* functions make your code easier to read and reason about.

Note: you can try this tutorial in .

2.7 Methods and functions

Now we know the basics of channels and operations on them, we will learn about additional methods and functions that can be convenient in various situations.

2.7.1 Putting and getting

As we have already seen, we can `add` into a channel. Immediately closing the channel afterwards ensures that no further items can be put into the channel:

```
In [2]: import aiochan as ac
        import asyncio

        async def main():
            c = ac.Chan().add(1, 2, 3).close()
            async for v in c:
                print(v)
            await c.put(4)
            r = await c.get()
            print('put/get after closing:', r)

        ac.run(main())
```

```
1
2
3
put/get after closing: None
```

This method is mainly provided for convenience. You should NOT adding too much stuff into a channel in this way: it is non-blocking, the puts are accumulated, and if there are too many pending puts accumulated in this way overflow will occur. Adding fewer than 10 items during the initialization phase of a channel is considered ok though.

In the last example we consumed values using the `async for` syntax. In case where we *must* deal with many values of the channel at once instead of one by one, we can use `collect`:

```
In [3]: async def main():
            c = ac.Chan().add(1, 2, 3).close()
            r = await c.collect()
            print(r)

        ac.run(main())
```

```
[1, 2, 3]
```

In this case, closing the channel first before calling `collect` is essential: otherwise the `await` would block forever (and overflow would probably occur if values continuously come in).

`collect` also accepts an argument `n` which specifies the maximum number of elements that will be collected. Using it, we can `collect` on channels that are not yet closed (but we still need to think about how many items we can deal with):

```
In [4]: async def main():
            c = ac.Chan().add(1, 2, 3) # no closing
            r = await c.collect(2)
            print(r)

        ac.run(main())
```

```
[1, 2]
```

Above we have said that using `add` to add too many items is dangerous. If you have an existing sequence which you want to turn into a channel, it is much better to use `from_iter`:

```
In [5]: async def main():
            c = ac.from_iter([1, 2, 3, 4, 5, 6])
            r = await c.collect()
            print(r)
            print(c.closed)

        ac.run(main())
```

```
[1, 2, 3, 4, 5, 6]
True
```

Note that the channel is closed on construction (we can check whether a channel is closed by using the `.closed` property on a channel).

Infinite collections are ok:

```
In [6]: def natural_numbers():
            i = 0
            while True:
                yield i
                i += 1

        async def main():
            c = ac.from_iter(natural_numbers())
            r = await c.collect(10)
            print(r)
            print(c.closed)
            r = await c.collect(10)
            print(r)
            print(c.closed)

        ac.run(main())
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
True
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
True
```

Even when the channel is closed, values can still be obtained from it (and in this case values cannot be exhausted). Closing only stops putting operations immediately.

Making channels producing numbers is so common that we have a function for it:

```
In [7]: async def main():
            c1 = ac.from_range()
            r = await c1.collect(10)
            print(r) # natural numbers

            c2 = ac.from_range(5) # same as ac.from_iter(range(5))
            r = await c2.collect()
            print(r)

            c3 = ac.from_range(0, 10, 3) # same as ac.from_iter(range(0, 10, 3))
            r = await c3.collect()
            print(r)

        ac.run(main())
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4]
```

```
[0, 3, 6, 9]
```

To recap:

- `.add` can be used to add a few items into a channel on initialization (any other use is dangerous)
- `.collect` can be used to bulk get items from channel
- `.closed` tests if a channel is already closed
- `from_iter` creates channels containing all elements from an iterable (even infinite iterable is ok)
- `from_range` is tailored for making channels generating number series

2.7.2 Time-based operations

So far we have always used `asyncio.sleep` to make execution stop for a little while, pretending to do work. We also have `timeout` function that does almost the same thing by producing a channel that automatically closes after an interval:

```
In [8]: async def main():
            start = asyncio.get_event_loop().time()
            c = ac.timeout(1.0)
            await c.get()
            end = asyncio.get_event_loop().time()
            print(end - start)

        ac.run(main())
```

```
1.001395168947056
```

This is useful even when we are not pretending to do work, for example, for timeout control:

```
In [9]: async def main():
            tout = ac.timeout(1.0)
            while (await ac.select(tout, default=True))[0]:
                print('do work')
                await asyncio.sleep(0.2)
            print('done')

        ac.run(main())
```

```
do work
do work
do work
do work
do work
done
```

The above example is written in a somewhat terse style. You should try to understand why it achieves the closing on time behaviour.

As `timeout` produces a channel, which can be passed arount and `select`ed, it offers great flexibility for controlling time-based behaviours. However, using it for the ticks of a clock is harmful, as exemplified below:

```
In [10]: async def main():
             start = asyncio.get_event_loop().time()
             for i in range(20):
                 await ac.timeout(0.1).get()
                 print(i, asyncio.get_event_loop().time() - start)

         ac.run(main())
```

```
0 0.10043401701841503
1 0.20142484991811216
2 0.30242938199080527
3 0.4030482260277495
4 0.5035843959776685
5 0.6041081629227847
6 0.7046528200153261
7 0.8056348919635639
8 0.9063465989893302
9 1.0068686519516632
10 1.1073921599891037
11 1.2079381300136447
12 1.3089604979613796
13 1.4095268349628896
14 1.5100650689564645
15 1.6105891889892519
16 1.7114433919778094
17 1.81249319401104
18 1.9130375039530918
19 2.0135989299742505
```

The problem is that `timeout` guarantees that it will close *after* the specified time has elapsed, and will make an attempt to close as soon as possible, but it can never close at the precise instant. Over time, errors will accumulate. In the above example, we have already accumulated 0.01 seconds of error in mere 2 seconds.

If want something that ticks, use the `tick_tock` function:

```
In [11]: async def main():
             start = asyncio.get_event_loop().time()
             ticker = ac.tick_tock(0.1)
             for i in range(20):
                 await ticker.get()
                 print(i, asyncio.get_event_loop().time() - start)

         ac.run(main())
```

```
0 0.1004815329797566
1 0.2012625669594854
2 0.3008053069934249
3 0.4013087539933622
4 0.5008452819893137
5 0.6013440380338579
6 0.7008649010676891
7 0.8013983579585329
8 0.900891529978253
9 1.001404833048582
10 1.100898704957217
11 1.2013944609789178
12 1.3008839710382745
13 1.4013996929861605
14 1.501174372038804
15 1.6006878040498123
16 1.701174663961865
17 1.8006792459636927
18 1.9011599159566686
19 2.000674612005241
```

Errors are still unavoidable, but they do not accumulate.

To recap:

- Use `timeout` to control the timing of operations (maybe together with `select`)
- If the timing control is recurrent, consider using `tick_tock`

2.7.3 Functional methods

If you have done any functional programming, you are certainly familiar with things like `map`, `reduce` (or `foldl`, `foldr`), `filter` and friends. Channels are armed with these so-called functional chainable methods which, when called, return new channels containing the expected elements.

Examples:

```
In [12]: async def main():
             print('map', await ac.from_range(10).map(lambda x: x*2).collect())
             print('filter', await ac.from_range(10).filter(lambda x: x % 2 == 0).collect())
             print('take', await ac.from_range(10).take(5).collect())
             print('drop', await ac.from_range(10).drop(5).collect())
             print('take_while', await ac.from_range(10).take_while(lambda x: x < 5).collect())
             print('drop_while', await ac.from_range(10).drop_while(lambda x: x < 5).collect())

         ac.run(main())
```

```
map [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
filter [0, 2, 4, 6, 8]
take [0, 1, 2, 3, 4]
drop [5, 6, 7, 8, 9]
take_while [0, 1, 2, 3, 4]
drop_while [5, 6, 7, 8, 9]
```

There is also `distinct`:

```
In [13]: async def main():
             c = ac.from_iter([0,0,0,1,1,2,2,2,2,3,3,4,4,4,5,4,4,3,3,2,1,1,1,0])
             print(await c.distinct().collect())

         ac.run(main())
```

```
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
```

Note that only *consecutive* values are tested for distinctness.

You probably know `reduce`, the so-called universal reducing function:

```
In [14]: async def main():
             print(await ac.from_range(10).reduce(lambda a, b: a+b).collect())
             print(await ac.from_range(10).reduce(lambda acc, nxt: acc + [nxt], init=[]).collect())

         ac.run(main())
```

```
[45]
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

As we can see, you can optionally pass an initial value for `reduce`. Notice that `reduce` only returns a value when the channel is closed: it turns a whole channel of values into a channel containing only a single value. Most of the time you may want intermediate results as well, so you probably want to use `scan` instead:

```
In [15]: async def main():
             print(await ac.from_range(10).scan(lambda a, b: a+b).collect())
             print(await ac.from_range(10).scan(lambda acc, nxt: acc + [nxt], init=[]).collect())

         ac.run(main())
```

```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
[[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5], [0, 1, 2, 3, 4, 5, 6
```

All of these "functional" methods accept two optional values: `out` and `close`. As we have said previously, these functions operate by returning a new channel containing the processed values. If another channel is given as the `out` argument, then that channel will receive the processed values instead. Also, when the source channel is closed, by default the out channel will be as well. You can prevent this by setting `close` to `False`. This is illustrated below:

```
In [16]: async def main():
             out = ac.Chan(5) # we can use buffers as we please
             ac.from_range(10).map(lambda x: x*2, out=out, close=False)
             print(out.closed)
             print(await out.collect(10))

         ac.run(main())
```

```
False
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

To recap:

- `map`, `reduce`, `filter`, `distinct`, `take`, `drop`, `take_while`, `drop_while`, `scan` do what you expect them to do.
- You can control the construction of the output channel and whether to close it when the input is exhausted by specifying the `out` and `close` argument.

2.7.4 Pipeline methods

There are times that your processing is rather complicated to express with the above functional methods. For example, given the sequence `[1,2,1,3,1]`, you want to produce the sequence `[1,2,2,1,3,3,3,1]`. In this case you can use the `async_apply` method:

```
In [17]: async def duplicate_face_value(inp, out):
             async for v in inp:
                 for _ in range(v):
                     await out.put(v)
             out.close()

         async def main():
             vals = [1,2,3,2,1]
             print(await ac.from_iter(vals).async_apply(duplicate_face_value).collect())

         ac.run(main())
```

```
[1, 2, 2, 3, 3, 3, 2, 2, 1]
```

You may think that this is not too different from connecting the channels yourself and spawn a processing coroutine with `go`. But writing it using `async_apply` makes your intention clearer.

Processing values in a channel and putting the result onto another channel is a very common theme. With `async_apply`, only a single coroutine is working on the values. With `async_pipe`, you can use multiple coroutine instances, getting closer to parallelism:

```
In [18]: async def worker(n):
             await asyncio.sleep(0.1)
             return n*2

         async def main():
             start = asyncio.get_event_loop().time()
             print(await ac.from_range(20).async_pipe(10, worker).collect())
```

```
        print(asyncio.get_event_loop().time() - start)

    ac.run(main())
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
0.20754481800395297
```

We see that processing 20 values only takes about 0.2 seconds even though processing a single value with a single coroutine takes 0.1 seconds: parallelism.

Notice that the output values are in the correct order. This is the case even if later works complete earlier: `async_pipe` ensures the order while doing its best to have the minimal waiting time. However, in some cases the order is not important, in which case we can use `async_pipe_unordered`:

```
In [19]: import random

    async def worker(n):
        await asyncio.sleep(random.uniform(0, 0.2))
        return n*2

    async def main():
        start = asyncio.get_event_loop().time()
        print(await ac.from_range(20).async_pipe(10, worker).collect())
        print('ordered time:', asyncio.get_event_loop().time() - start)

        start = asyncio.get_event_loop().time()
        print(await ac.from_range(20).async_pipe_unordered(10, worker).collect())
        print('unordered time:', asyncio.get_event_loop().time() - start)

    ac.run(main())
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
ordered time: 0.33254893589764833
[14, 8, 6, 0, 20, 12, 24, 10, 22, 16, 36, 18, 2, 4, 26, 28, 38, 30, 32, 34]
unordered time: 0.2875210080528632
```

We see that unordered processing in the face of random processing time has efficiency advantage.

To recap:

- use `async_apply` to give your custom processing pipeline a uniform look
- use `async_pipe` for parallelism within asyncio
- you can get more concurrency with `async_pipe_unordered`, but you give up the return order

Note: you can try this tutorial in .

2.8 Combination operations

Channels can act as versatile conduits for the flow of data, as in our examples of the fan-in and fan-out pattern. Here we discuss some convenient functions and constructs for dealing with more complicated patterns for the combination of data.

2.8.1 Merging values

In fan-in, we passed the channels to all coroutines interested in producing data. Often we find that these producers provide their own output channels instead. In this case, we can `merge` these channels into a single one:

```
In [2]: import aiochan as ac
        import asyncio

        async def main():
            p1 = ac.from_range(10).map(lambda x: 'p1-' + str(x))
            p2 = ac.from_range(10).map(lambda x: 'p2-' + str(x))
            out = ac.merge(p1, p2)
            async for v in out:
                print(v)

        ac.run(main())
```

```
p2-0
p1-0
p2-1
p1-1
p2-2
p1-2
p2-3
p1-3
p2-4
p1-4
p1-5
p2-5
p1-6
p2-6
p2-7
p1-7
p2-8
p1-8
p2-9
p1-9
```

As in manual fan-in, the order of the values are somewhat non-deterministic. If you want your channels to produce values in-sync, you want to use `zip_chans`:

```
In [3]: async def main():
            p1 = ac.from_range(10).map(lambda x: 'p1-' + str(x))
            p2 = ac.from_range(10).map(lambda x: 'p2-' + str(x))
            out = ac.zip_chans(p1, p2)
            async for v in out:
                print(v)

        ac.run(main())
```

```
['p1-0', 'p2-0']
['p1-1', 'p2-1']
['p1-2', 'p2-2']
['p1-3', 'p2-3']
['p1-4', 'p2-4']
['p1-5', 'p2-5']
['p1-6', 'p2-6']
['p1-7', 'p2-7']
['p1-8', 'p2-8']
['p1-9', 'p2-9']
```

A even more complicated use case is that your producer produces items at different rates, and you want to keep track of the *latest* values produced by each of them:

```
In [4]: async def main():
            p1 = ac.from_range(10).map(lambda x: 'p1-' + str(x))
            p2 = ac.from_range(10).map(lambda x: 'p2-' + str(x))
            out = ac.combine_latest(p1, p2)
            async for v in out:
                print(v)

        ac.run(main())
```

```
['p1-0', None]
['p1-0', 'p2-0']
['p1-1', 'p2-0']
['p1-1', 'p2-1']
['p1-1', 'p2-2']
['p1-2', 'p2-2']
['p1-2', 'p2-3']
['p1-3', 'p2-3']
['p1-4', 'p2-3']
['p1-4', 'p2-4']
['p1-5', 'p2-4']
['p1-5', 'p2-5']
['p1-6', 'p2-5']
['p1-6', 'p2-6']
['p1-7', 'p2-6']
['p1-7', 'p2-7']
['p1-7', 'p2-8']
['p1-8', 'p2-8']
['p1-9', 'p2-8']
['p1-9', 'p2-9']
```

Notice for channels that has yet to produce a value, `None` is put in its place.

As for the functional methods, all of these functions take optional `out` and `close` arguments, controlling the output channel and whether to close the output channel when there is nothing more to be done (i.e., when all source channels are closed).

To recap:

- use `merge` to combine values from several channels
- use `zip_chans` to combine values from several channels in sync
- use `combine_latest` to combine values and monitor the latest value from each channel

2.8.2 Distributing values

We have discussed generalizations of fan-in above. For simple fan-out, we have `distribute`:

```
In [5]: async def worker(inp, tag):
            async for v in inp:
                print('%s received %s' % (tag, v))

        async def main():
            inputs = [ac.Chan(name='inp%s' % i) for i in range(3)]
            ac.from_range(20).distribute(*inputs)
            for idx, c in enumerate(inputs):
                ac.go(worker(c, 'worker%s' % idx))
            await asyncio.sleep(0.1)

        ac.run(main())
```

```
worker0 received 0
worker0 received 1
```

```
worker0 received 4
worker1 received 2
worker2 received 3
worker0 received 5
worker0 received 8
worker2 received 6
worker1 received 7
worker0 received 9
worker0 received 12
worker1 received 10
worker2 received 11
worker1 received 13
worker1 received 16
worker0 received 14
worker2 received 15
worker1 received 17
worker2 received 18
worker0 received 19
```

One of the benefit of using `distribute` instead of a plain fan-out is that, in the case one of the down-stream channels are closed, `distribute` will try to put the value into another downstream channel so that no values would be lost.

In fan-out and `distribute`, each down-stream consumer obtains non-overlapping subsets of the input. Sometimes we want each consumer to consume the whole input instead. In this case, we want a duplicator, or `Dup` in aiochan. An example:

```
In [6]: async def worker(inp, tag):
            async for v in inp:
                print('%s received %s' % (tag, v))

        async def main():
            dup = ac.from_range(5).dup()
            inputs = [dup.tap() for i in range(3)]

            for idx, c in enumerate(inputs):
                ac.go(worker(c, 'worker%s' % idx))
            await asyncio.sleep(0.1)
            dup.close()

        ac.run(main())
```

```
worker0 received 0
worker1 received 0
worker1 received 1
worker2 received 0
worker0 received 1
worker2 received 1
worker2 received 2
worker0 received 2
worker1 received 2
worker0 received 3
worker0 received 4
worker1 received 3
worker2 received 3
worker1 received 4
worker2 received 4
```

We see that `ch.dup` creates a duplicator, or `Dup`, and `dup.tap()` creates a new tap on the duplicator that contains the values put into the source channel. As for the functional methods, `dup.tap` accepts the arguments `out` and `close`, which controls what to be used as the output channel and whether to close the output channel when the input is closed.

Note that duplicated elements are put to downstream channels in order. This means that if any one of the downstream channels block on put for some reason, the whole progress will be blocked. You should consider giving downstream inputs some buffer if your downstream processors are uneven in their processing speed.

A `Dup` also has the method `untap`, which can be used to untap an existing tapping channel. For example:

```
In [7]: async def worker(inp, tag):
            async for v in inp:
                print('%s received %s' % (tag, v))

        async def main():
            dup = ac.from_range(5).dup()
            inputs = [dup.tap() for i in range(3)]
            dup.untap(inputs[1])

            for idx, c in enumerate(inputs):
                ac.go(worker(c, 'worker%s' % idx))
            await asyncio.sleep(0.1)
            dup.close()

        ac.run(main())
```

```
worker0 received 0
worker2 received 0
worker2 received 1
worker0 received 1
worker0 received 2
worker0 received 3
worker2 received 2
worker2 received 3
worker2 received 4
worker0 received 4
```

We see that `worker1`, which has been untapped, does not receive anything. In more complicated programs, tappings and untapping can be done dynamically, at arbitrary times.

Another very common idiom is pub-sub, and this is easy to do as well:

```
In [9]: async def processor(inp, tag):
            async for v in inp:
                print('%s received %s' % (tag, v))

        async def main():
            source = ac.Chan()
            pub = source.pub(lambda x: x % 3)
            p1 = pub.sub(1)
            p2 = pub.sub(2)
            p0 = pub.sub(0)
            px = pub.sub(0)
            ac.go(processor(p1, 'p1'))
            ac.go(processor(p2, 'p2'))
            ac.go(processor(p0, 'p0'))
            ac.go(processor(px, 'px'))
            source.add(0,1,2,3,4,5,6,7,8,9).close()
            await asyncio.sleep(0.1)
            pub.close()

        ac.run(main())
```

```
p0 received 0
```

```
px received 0
px received 3
p0 received 3
p1 received 1
p2 received 2
p0 received 6
px received 6
p1 received 4
p2 received 5
p1 received 7
p2 received 8
p0 received 9
px received 9
```

In this case, the topic is defined by the lambda, which gives the remainder when the item is divided by three. Processors subscribe to the topics they are intrested in, and we see that `p0` and `px` received all numbers with remainder 0, `p1` all numbers with remainder 1, and `p2` all numbers with remainder 2.

A `Pub` also has a method `unsub`, which can be used to unsubscribe a currently subscribing channel. For example:

```
In [11]: async def processor(inp, tag):
             async for v in inp:
                 print('%s received %s' % (tag, v))

         async def main():
             source = ac.Chan()
             pub = source.pub(lambda x: x % 3)
             p1 = pub.sub(1)
             p2 = pub.sub(2)
             p0 = pub.sub(0)
             px = pub.sub(0)
             ac.go(processor(p1, 'p1'))
             ac.go(processor(p2, 'p2'))
             ac.go(processor(p0, 'p0'))
             ac.go(processor(px, 'px'))
             pub.unsub(0, px)
             pub.sub(1, px)
             pub.sub(2, px)
             source.add(0,1,2,3,4,5,6,7,8,9).close()
             await asyncio.sleep(0.1)
             pub.close()

         ac.run(main())
```

```
p0 received 0
p0 received 3
p1 received 1
px received 1
px received 2
p2 received 2
p1 received 4
px received 4
px received 5
p2 received 5
p2 received 8
px received 8
px received 7
p0 received 6
p1 received 7
p0 received 9
```

In this example, we initially subscribed `px` to the topic 0, but then changed our mind and subscribed it to 1 and 2 instead (yes a channel can subscribe to multiple topics).

There is also `unsub_all`, which can unsubscribe a whole topic in one go:

```
In [13]: async def processor(inp, tag):
             async for v in inp:
                 print('%s received %s' % (tag, v))

         async def main():
             source = ac.Chan()
             pub = source.pub(lambda x: x % 3)
             p1 = pub.sub(1)
             p2 = pub.sub(2)
             p0 = pub.sub(0)
             px = pub.sub(0)
             ac.go(processor(p1, 'p1'))
             ac.go(processor(p2, 'p2'))
             ac.go(processor(p0, 'p0'))
             ac.go(processor(px, 'px'))
             pub.unsub_all(0)
             source.add(0,1,2,3,4,5,6,7,8,9)
             await asyncio.sleep(0.1)
             pub.close()

         ac.run(main())
```

```
p1 received 1
p1 received 4
p2 received 2
p2 received 5
p2 received 8
p1 received 7
```

Now the 0 topic does not have any subscribers after the call to `unsub_all`. If this method is called without argument, *all* subscribers for *all* topics are unsubscribed.

In the above examples, we have passed in a lambda for the argument to `pub`. If we don't pass in anything, then the `Pub` assumes that values are tuples, and the first element of the tuple is the topic:

```
In [15]: async def processor(inp, tag):
             async for v in inp:
                 print('%s received %s' % (tag, v))

         async def main():
             source = ac.Chan()
             pub = source.pub()
             p1 = pub.sub(1)
             p2 = pub.sub(2)
             p0 = pub.sub(0)
             px = pub.sub(0)
             ac.go(processor(p1, 'p1'))
             ac.go(processor(p2, 'p2'))
             ac.go(processor(p0, 'p0'))
             ac.go(processor(px, 'px'))
             source.add((0, 0),
                        (1, 1),
                        (2, 2),
                        (0, 3),
                        (1, 4),
                        (2, 5),
                       (0, 6),
                       (1, 7),
                       (2, 8),
                       (3, 9))
             await asyncio.sleep(0.1)
             pub.close()

         ac.run(main())
```

```
p0 received (0, 0)
px received (0, 0)
px received (0, 3)
p0 received (0, 3)
p1 received (1, 1)
p2 received (2, 2)
p0 received (0, 6)
px received (0, 6)
p1 received (1, 4)
p2 received (2, 5)
p1 received (1, 7)
p2 received (2, 8)
```

As before, `sub` methods all take `out` and `close` arguments that have their usual meaning.

To recap:

- use `distribute` to distribute values to downstream channels
- use `dup` to duplicate values
- use `pub` for publisher-subscriber systems

Note: you can try this tutorial in .

2.9 Parallelism and beyond

We discussed `async_pipe` and `async_pipe_unordered` in the context of trying to put more "concurrency" into our program by taking advantage of parallelism. What does "parallelism" mean here?

2.9.1 Facing the reality of python concurrency, again

With `async_pipe` and `async_pipe_unordered`, by giving them more coroutine instances to work with, we achieved higher throughput. But that is only because our coroutines are, in a quite literal sense, sleeping on the job: to simulate real jobs, we called `await` on `asyncio.sleep`. The event loop, faced with this await, just puts the coroutine on hold until it is ready to act again.

Now it is entirely possible that this behaviour — of not letting sleeping coroutines block the whole program — is all you need. In particular, if you are dealing with network connections or sockets *and* you are using a proper asyncio-based library, then "doing network work" isn't too much from sleeping on the loop.

However, for other operations *not* tailored for asyncio, you will *not* get any speed-up with parallelism based on asyncio. Crucially, *asyncio has no built-in support for file accesses*.

Let's see an example:

```
In [2]: import asyncio
        import time
        import aiochan as ac
        async def worker(n):
            time.sleep(0.1) # await asyncio.sleep(0.1)
            return n*2

        async def main():
            start = asyncio.get_event_loop().time()
            print(await ac.from_range(20).async_pipe(10, worker).collect())
            print(asyncio.get_event_loop().time() - start)

        ac.run(main())
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
2.009612141060643
```

The only different than before (when we first introduced `async_pipe`) is that we replaced `asyncio.sleep` with `time.sleep`. With this change, we did not get *any* speed up.

In this case, we can recover our speed-up by using the method `parallel_pipe` instead:

```
In [3]: import asyncio
        import time
        import aiochan as ac

        def worker(n):
            time.sleep(0.1)
            return n*2

        async def main():
            start = asyncio.get_event_loop().time()
            print(await ac.from_range(20).parallel_pipe(10, worker).collect())
            print(asyncio.get_event_loop().time() - start)

        ac.run(main())
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
0.20713990507647395
```

When using `parallel_pipe`, our `worker` has to be a normal function instead of an async function. As before, if order is not important, `parallel_pipe_unordered` can give you even more throughput:

```
In [5]: import asyncio
        import time
        import random
        import aiochan as ac

        def worker(n):
            time.sleep(random.uniform(0, 0.2))
            return n*2

        async def main():
            start = asyncio.get_event_loop().time()
            print(await ac.from_range(20).parallel_pipe(10, worker).collect())
            print('ordered time:', asyncio.get_event_loop().time() - start)

            start = asyncio.get_event_loop().time()
            print(await ac.from_range(20).parallel_pipe_unordered(10, worker).collect())
            print('unordered time:', asyncio.get_event_loop().time() - start)

        ac.run(main())
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
ordered time: 0.35387236496899277
```

```
[16, 2, 8, 24, 6, 10, 0, 32, 22, 34, 12, 36, 4, 38, 28, 18, 30, 20, 14, 26]
unordered time: 0.19887939398176968
```

In fact, `parallel_pipe` works by starting a thread-pool and execute the workers in the thread-pool. Multiple threads can solve the problem of workers sleeping on the thread, as in our example. But remember that the default implementation of python, the CPython, has a global interpreter lock (GIL) which prevents more than one python statement executing at the same time. Will `parallel_pipe` help in the presence of GIL, besides the case of workers just sleeping?

It turns out that for the majority of serious cases, multiple threads help even in the presence of the GIL, because most of the heavy-lifting operations, for example file accesses, are implemented in C instead of in pure python, and in C it is possible to release the GIL when not interacting with the python runtime. In addition to file accesses, if you are doing number-crunching, then hopefully you are not doing it in pure python but instead relies on dedicated libraries like numpy, scipy, etc. All of these libraries release the GIL when it makes sense to do so. So using `parallel_pipe` is usually enough.

What if you just have to do your CPU-intensive tasks in python? Well, `parallel_pipe` and `parallel_pipe_unordered` takes an argument called `mode`, which by default takes the value `'thread'`. If you change it to `'process'`, then a process-pool instead of a thread-pool will be used. Let's see a comparison:

```
In [6]: import asyncio
        import time
        import aiochan as ac

        def worker(_):
            total = 0
            for i in range(1000000):
                total += i
            return total

        async def main():
            start = asyncio.get_event_loop().time()
            await ac.from_range(20).parallel_pipe(10, worker).collect()
            print('using threads', asyncio.get_event_loop().time() - start)

            start = asyncio.get_event_loop().time()
            await ac.from_range(20).parallel_pipe(10, worker, mode='process').collect()
            print('using threads', asyncio.get_event_loop().time() - start)

        ac.run(main())
```

```
using threads 1.7299788249656558
using threads 0.20847543003037572
```

Why not use a process pool in all cases? Processes have much greater overhead than threads, and also far more restrictions on their use. Crucially, you cannot share any object unless you do some dirty work yourself, and anything you pass to your worker, or return from your worker, must be picklable.

In our example, our worker is a pure function. It is also possible to prepare some structures in each worker before-hand. In python 3.7 or above, there are the `initializer` and `init_args` arguments accepted by `parallel_pipe` and `parallel_pipe_unordered`, which will be passed to the construction to the pool executors to do the setup. Prior to python 3.7, such a setup is still possible with some hack: you can put the object to be set up in a `threading.local` object, and for *every* worker execution, check if the object exists, and if not, do the initialization:

```
In [7]: import asyncio
        import time
        import random
        import threading
        import aiochan as ac
        worker_data = threading.local()

        def worker(n):
            try:
                processor = worker_data.processor
            except:
                print('setting up processor')
                worker_data.processor = lambda x: x*2
                processor = worker_data.processor
            return processor(n)

        async def main():
            start = asyncio.get_event_loop().time()
            print(await ac.from_range(20).parallel_pipe(2, worker).collect())

        ac.run(main())
```

```
setting up processor
setting up processor
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

Since we used two thread workers, the setup is done twice. This also works for `mode='process'`.

What about parallelising work across the network? Or more exotic workflows? At its core, *aiochan* is a library that facilitates moving data around within the boundary of a single process on a single machine, but there is nothing preventing you using channels at the end-points of a network-based parallelism framework such as message queues or a framework like *dart*. Within its bounday, *aiochan* aims to give you maximum flexibility in developing concurrent workflows, and you should use *aiochan* it in tandem with some other suitable libraries or frameworks when you want to step out of its boundary.

2.9.2 Back to the main thread

Speaking of stepping out of boundaries, one case is exceedingly common: you use an aiochan-based workflow to prepare a stream of values, but you want to consume these values outside of the asyncio event loop. In this case, there are convenience methods for you:

```
In [8]: loop = asyncio.new_event_loop()

        out = ac.Chan(loop=loop)

        async def worker():
            while True:
                await asyncio.sleep(0.1)
                if not (await out.put('work')):
                    break

        ac.run_in_thread(worker(), loop=loop)

        it = out.to_iterable(buffer_size=1)

        print(next(it))
        print(next(it))

        loop.call_soon_threadsafe(out.close);
```

```
work
work
```

Notice how we constructed the channel on the main thread, with explicit arguments specifying on which loop the channel is to be used, and then derived a iterator from the queue. Also, to run the worker, we used `run_in_thread` with an explicit event loop given.

When creating the iterable, notice we have given it a `buffer_size`. This is used to construct a queue for inter-thread communication. You can also use a queue directly:

```
In [9]: import queue

        loop = asyncio.new_event_loop()

        out = ac.Chan(loop=loop)

        async def worker():
            while True:
                await asyncio.sleep(0.1)
                if not (await out.put('work')):
                    break

        ac.run_in_thread(worker(), loop=loop)

        q = queue.Queue()

        out.to_queue(q)

        print(q.get())
        print(q.get())

        loop.call_soon_threadsafe(out.close);
```

```
work
work
```

Other queues can be used as long as they follow the public API of `queue.Queue` and are thread-safe.

2.9.3 aiochan without asyncio

Finally, before ending this tutorial, let's reveal a secret: you don't need asyncio to use aiochan! "Isn't aiochan based on asyncio?" Well, not really, the core algorithms of aiochan (which is based on those from Clojure's core.async) does not use any asyncio constructs: they run entirely synchronously. It is only when you use the use-facing methods such as `get`, `put` and `select` that an asyncio-facade was made to cover the internals.

On the other hand, there are some functions (actually, three of them) that does not touch anything related to asyncio given the correct arguments:

- `Chan.put_nowait`
- `Chan.get_nowait`
- `select`

Normally, when you call `ch.put_nowait(v)`, the put will succeed if it is possible to do so immediately (for example, if there is a pending get or buffer can be used), otherwise it will give up. Note that you never `await` on `put_nowait`. However, if you give the argument `immediate_only=True`, then if the operation cannot be completed immediately, it will be queued (but again, the pending queue can overflow). In addition, you can give a callback to the `cb` argument, which will be called when the put finally succeeds, with the same argument as the return value of `await put(v)`. The same is true with `get_nowait(immediate_only=True, cb=cb)`. For `select`, if you give a callback to the `cb` argument, then you should not call `await` on it, but instead rely on the callback being called eventually as `cb(return_value, which_channel)`. Note if you don't expect to use any event loops, when constructing channels, you should explicitly pass in `loop='no_loop'`.

Example: this is our asyncio-based fan-in, fan-out:

```
In [10]: import aiochan as ac
         import asyncio

         async def consumer(c, tag):
             async for v in c:
                 print('%s received %s' % (tag, v))

         async def producer(c, tag):
             for i in range(5):
                 v = '%s-%s' % (tag, i)
                 print('%s produces %s' % (tag, v))
                 await c.put(v)

         async def main():
             c = ac.Chan()
             for i in range(3):
                 ac.go(consumer(c, 'c' + str(i)))
             for i in range(3):
                 ac.go(producer(c, 'p' + str(i)))
             await asyncio.sleep(0.1)

         ac.run(main())
```

```
p0 produces p0-0
p0 produces p0-1
p0 produces p0-2
p0 produces p0-3
p1 produces p1-0
p2 produces p2-0
c0 received p0-0
c0 received p0-3
c0 received p1-0
c0 received p2-0
c1 received p0-1
c2 received p0-2
p0 produces p0-4
p1 produces p1-1
p1 produces p1-2
p1 produces p1-3
p2 produces p2-1
c0 received p0-4
c0 received p1-3
c0 received p2-1
c1 received p1-1
c2 received p1-2
p1 produces p1-4
p2 produces p2-2
p2 produces p2-3
p2 produces p2-4
c0 received p1-4
c0 received p2-4
c1 received p2-2
c2 received p2-3
```

By the appropriate use of callbacks, we can write avoid using `asyncio` completely:

```
In [12]: def consumer(c, tag):
             def cb(v):
                 if v is not None:
                     print('%s received %s' % (tag, v))
                     consumer(c, tag)
             c.get_nowait(immediate_only=False, cb=cb)

         def producer(c, tag, i=0):
             v = '%s-%s' % (tag, i)
             def cb(ok):
                 if ok and i < 4:
                     print('%s produces %s' % (tag, v))
                     producer(c, tag, i+1)

             c.put_nowait(v, immediate_only=False, cb=cb)

         def main():
             c = ac.Chan(loop='no_loop')
             for i in range(3):
                 consumer(c, 'c' + str(i))
             for i in range(3):
                 producer(c, 'p' + str(i))

         main()
```

```
c0 received p0-0
p0 produces p0-0
c1 received p0-1
p0 produces p0-1
c2 received p0-2
p0 produces p0-2
c0 received p0-3
p0 produces p0-3
c1 received p0-4
c2 received p1-0
p1 produces p1-0
c0 received p1-1
p1 produces p1-1
c1 received p1-2
p1 produces p1-2
c2 received p1-3
p1 produces p1-3
c0 received p1-4
c1 received p2-0
p2 produces p2-0
c2 received p2-1
p2 produces p2-1
c0 received p2-2
p2 produces p2-2
c1 received p2-3
p2 produces p2-3
c2 received p2-4
```

The end result is (almost) the same. An example with `select`:

```
In [13]: def select_run():
             c = ac.Chan(1, loop='no_loop', name='c')
             d = ac.Chan(1, loop='no_loop', name='d')
             put_chan = None

             def put_cb(v, c):
                 nonlocal put_chan
```

```
                put_chan = c

            ac.select((c, 1), (d, 2), cb=put_cb)

            get_val = None

            def get_cb(v, c):
                nonlocal get_val
                get_val = v

            ac.select(c, d, cb=get_cb)

            print('select put into %s, get value %s' % (put_chan, get_val))

        def main():
            for _ in range(10):
                select_run()

        main()
```

```
select put into Chan<c 140356982933192>, get value 1
select put into Chan<c 140356982933192>, get value 1
select put into Chan<d 140356982931944>, get value 2
select put into Chan<c 140356982931944>, get value 1
select put into Chan<c 140356982931944>, get value 1
select put into Chan<c 140356982931944>, get value 1
select put into Chan<d 140356982932672>, get value 2
select put into Chan<c 140356982932672>, get value 1
select put into Chan<d 140356982931944>, get value 2
select put into Chan<c 140356982931944>, get value 1
```

"But why?" Well, obviously writing callbacks is much harder than using asyncio. But who knows? Maybe you are writing some other, higher-level framework that can make use of the semantics of aiochan. The possibilities are endless! In particular, there are non-asyncio concurrency frameworks in python itself that utilizes the same coroutines, an example being `python-trio`. Since the core of aiochan does not rely on asyncio, porting it to `trio` is trivial.

CHAPTER 3

FAQ

Is aiochan original? Almost *nothing* in aiochan is original. It is not intended to be original.

The theory is of course laid out long ago, for example see the CSP paper and the Dijkstra paper.

The API is mostly inspired by Golang and Clojure's core.async, with a few pythonic touches here and there.

The implementation of the core logic of channels is mostly a faithful port of Clojure's core.async. But then you can have confidence in the correctness of results that aiochan gives, since core.async *is* battle-tested. Writing concurrency libraries *from scratch* is difficult, and writing *correct* ones *from scratch* takes many iterations and an unreasonable number of hours.

Aiochan *is* intended to be *unoriginal*, *useful*, *productive*, and *fun to use*.

If you like CSP, why don't you use *Golang/core.async/elixir/erlang* . . . directly? Well, we do use these things.

But often we find ourselves in the situation where python is the most natural language. However, without a high level concurrency control library, our code suffers from the following symptoms:

- brittle and impossible to understand due to ad-hoc non-deterministic callback hell;
- slow and idle running time due to single threaded code without any concurrent or parallel execution;
- slow and frustrating developing time due to drowning in locks, semaphores, etc.

Well, aiochan definitely cured these for us. And perhaps you will agree with me that porting core.async to python's asyncio is much easier than porting numpy/scipy/scikit-learn/pytorch/tensorflow . . . to some other language, or finding usable alternative thereof. Coercing libraries to behave well when unsuspectingly manipulated by a foreign language is agonizing.

CHAPTER 4

API Reference

All functions and classes from *aiochan.channel* can also be imported from the top-level `aiochan` module.

4.1 Channel

class aiochan.channel.**Chan**(*buffer=None*, *buffer_size=None*, *, *loop=None*, *name=None*)

A channel, the basic construct in CSP-style concurrency.

Channels can be used as async generators using the `async for` construct for async iteration of the values.

Parameters

- **buffer** – if a *aiochan.buffers.AbstractBuffer()* is given, then it will be used as the buffer. In this case *buffer_size* has no effect.

  If an integer is given, then a *aiochan.buffers.FixedLengthBuffer()* will be created with the integer value as the buffer size and used.

  If the a string value of *f*, *d*, *s* or *p* is given, a *aiochan.buffers.FixedLengthBuffer()*, *aiochan.buffers.DroppingBuffer()*, *aiochan.buffers.SlidingBuffer()* or *aiochan.buffers.PromiseBuffer()* will be created and used, with size given by the parameter *buffer_size*.

- **buffer_size** – see the doc for *buffer*.
- **loop** – the asyncio loop that should be used when scheduling and creating futures. If *None*, will use the current loop. If the special string value *"no_loop"* is given, then will not use a loop at all. Even in this case the channel can operate if you use only *aiochan.channel.Chan.get_nowait()* and *aiochan.channel.Chan.put_nowait()*.
- **name** – used to provide more friendly debugging outputs.

put(*val*)

Coroutine. Put a value into the channel.

Parameters **val** – value to put into the channel. Cannot be *None*.

Returns Awaitable of *True* if the op succeeds before the channel is closed, *False* if the op is applied to a then-closed channel.

put_nowait(*val*, *cb=None*, ***, *immediate_only=True*)

Put *val* into the channel synchronously.

If *immediate_only* is *True*, the operation will not be queued if it cannot complete immediately.

When *immediate_only* is *False*, *cb* can be optionally provided, which will be called when the put op eventually completes, with a single argument'True' or *False* depending on whether the channel is closed at the time of completion of the put op. *cb* cannot be supplied when *immediate_only* is *True*.

Returns *True* if the put succeeds immediately, *False* if the channel is already closed, *None* if the operation is queued.

add(**vals*)

Convenient method for putting many elements to the channel. The put semantics is the same as `aiochan.channel.Chan.put_nowait()` with *immediate_only=False*.

Note that this method can potentially overflow the channel's put queue, so it is only suitable for adding small number of elements.

Parameters `vals` – values to add, none of which can be *None*.

Returns *self*

get()

Coroutine. Get a value of of the channel.

Returns An awaitable holding the obtained value, or of *None* if the channel is closed before succeeding.

get_nowait(*cb=None*, ***, *immediate_only=True*)

try to get a value from the channel but do not wait. :type self: Chan :param self: :param cb: a callback to execute, passing in the eventual value of the get operation, which is None if the channel becomes closed before a value is available. Cannot be supplied when immediate_only is True. Note that if cb is supplied, it will be executed even when the value IS immediately available and returned by the function. :param immediate_only: do not queue the get operation if it cannot be completed immediately. :return: the value if available immediately, None otherwise

close()

Close the channel.

After this method is called, further puts to this channel will complete immediately without doing anything. Further gets will yield values in pending puts or buffer. After pending puts and buffer are both drained, gets will complete immediately with *None* as the result.

Closing an already closed channel is an no-op.

Returns *self*

closed

Returns whether this channel is already closed.

async_apply(*f=<function Chan._pipe_worker>*, *out=None*)

Apply a coroutine function to values in the channel, giving out an arbitrary number of results into the output channel and return the output value.

Parameters

- `out` – the *out* channel giving to the coroutine function *f*. If *None*, a new channel with no buffer will be created.

- **f** – a coroutine function taking two channels, *inp* and *out*. *inp* is the current channel and *out* is the given or newly created out channel. The coroutine function should take elements from *inp*, do its processing, and put the processed values into *out*. When, how often and whether values are put into *out*, and when or whether *out* is ever closed, is up to the coroutine.

  If *f* is not given, an identity coroutine function which will just pass the values along and close *out* when *inp* is closed is used.

Returns the *out* channel.

async_pipe (*n*, *f*, *out=None*, ***, *close=True*)
Asynchronously apply the coroutine function *f* to each value in the channel, and pipe the results to *out*. The results will be processed in unspecified order but will be piped into *out* in the order of their inputs.

If *f* involves slow or blocking operation, consider using *parallel_pipe*.

If ordering is not important, consider using *async_pipe_unordered*.

Parameters

- **n** – how many coroutines to spawn for processing.
- **f** – a coroutine function accepting one input value and returning one output value. S hould never return *None*.
- **out** – the output channel. if *None*, one without buffer will be created and used.
- **close** – whether to close the output channel when the input channel is closed.

Returns the output channel.

async_pipe_unordered (*n*, *f*, *out=None*, ***, *close=True*)
Asynchronously apply the coroutine function *f* to each value in the channel, and pipe the results to *out*. The results will be put into *out* in an unspecified order: whichever result completes first will be given first.

If *f* involves slow or blocking operation, consider using *parallel_pipe_unordered*.

If ordering is not important, consider using *async_pipe*.

Parameters

- **n** – how many coroutines to spawn for processing.
- **f** – a coroutine function accepting one input value and returning one output value. Should never return *None*.
- **out** – the output channel. if *None*, one without buffer will be created and used.
- **close** – whether to close the output channel when the input channel is closed.

Returns the output channel.

parallel_pipe (*n*, *f*, *out=None*, *mode='thread'*, *close=True*, ***kwargs*)
Apply the plain function *f* to each value in the channel, and pipe the results to *out*. The function *f* will be run in a pool executor with parallelism *n*. The results will be put into *out* in an unspecified order: whichever result completes first will be given first.

Note that even in the presence of GIL, *thread* mode is usually sufficient for achieving the greatest parallelism: the overhead is much lower than *process* mode, and many blocking or slow operations (e.g. file operations, network operations, *numpy* computations) actually release the GIL.

If *f* involves no blocking or slow operation, consider using *async_pipe_unordered*.

If ordering is important, consider using *parallel_pipe*.

Parameters

- **n** – the parallelism of the pool executor (number of threads or number of processes).
- **f** – a plain function accepting one input value and returning one output value. Should never return *None*.
- **out** – the output channel. if *None*, one without buffer will be created and used.
- **mode** – if *thread*, a *ThreadPoolExecutor* will be used; if *process*, a *ProcessPoolExecutor* will be used. Note that in the case of *process*, *f* should be a top-level function.
- **close** – whether to close the output channel when the input channel is closed.
- **kwargs** – theses will be given to the constructor of the pool executor.

Returns the output channel.

parallel_pipe_unordered(*n*, *f*, *out=None*, *mode='thread'*, *close=True*, ***kwargs*)

Apply the plain function *f* to each value in the channel, and pipe the results to *out*. The function *f* will be run in a pool executor with parallelism *n*. The results will be processed in unspecified order but will be piped into *out* in the order of their inputs.

Note that even in the presence of GIL, *thread* mode is usually sufficient for achieving the greatest parallelism: the overhead is much lower than *process* mode, and many blocking or slow operations (e.g. file operations, network operations, *numpy* computations) actually release the GIL.

If *f* involves no blocking or slow operation, consider using *async_pipe*.

If ordering is not important, consider using *parallel_pipe_unordered*.

Parameters

- **n** – the parallelism of the pool executor (number of threads or number of processes).
- **f** – a plain function accepting one input value and returning one output value. Should never return *None*.
- **out** – the output channel. if *None*, one without buffer will be created and used.
- **mode** – if *thread*, a *ThreadPoolExecutor* will be used; if *process*, a *ProcessPoolExecutor* will be used. Note that in the case of *process*, *f* should be a top-level function.
- **close** – whether to close the output channel when the input channel is closed.
- **kwargs** – theses will be given to the constructor of the pool executor.

Returns the output channel.

to_queue(*q*)

Put elements from the channel onto the given queue. Useful for inter-thread communication.

Parameters **q** – the queue.

Returns the queue *q*.

to_iterable(*buffer_size=1*)

Return an iterable containing the values in the channel.

This method is a convenience provided expressly for inter-thread usage. Typically, we will have an asyncio loop on a background thread producing values, and this method can be used as an escape hatch to transport the produced values back to the main thread.

If your workflow consists entirely of operations within the asyncio loop, you should use the channel as an async generator directly: `async for val in ch: ...`

This method should be called on the thread that attempts to use the values in the iterable, not on the thread on which operations involving the channel is run. The *loop* argument to the channel **must** be explicitly given, and should be the loop on which the channel is intended to be used.

Parameters **`buffer_size`** – buffering between the iterable and the channel.

Returns the iterable.

`map`(*f*, ***, *out=None*, *close=True*)
Returns a channel containing *f(v)* for values *v* from the channel.

Parameters

- **`close`** – whether *out* should be closed when there are no more values to be produced.
- **`out`** – the output channel. If *None*, one with no buffering will be created.
- **`f`** – a function receiving one element and returning one element. Cannot return *None*.

Returns the output channel.

`filter`(*p*, ***, *out=None*, *close=True*)
Returns a channel containing values *v* from the channel for which *p(v)* is true.

Parameters

- **`close`** – whether *out* should be closed when there are no more values to be produced.
- **`out`** – the output channel. If *None*, one with no buffering will be created.
- **`p`** – a function receiving one element and returning whether this value should be kept.

Returns the output channel.

`take`(*n*, ***, *out=None*, *close=True*)
Returns a channel containing at most *n* values from the channel.

Parameters

- **`n`** – how many values to take.
- **`out`** – the output channel. If *None*, one with no buffering will be created.
- **`close`** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

`drop`(*n*, ***, *out=None*, *close=True*)
Returns a channel containing values from the channel except the first *n* values.

Parameters

- **`n`** – how many values to take.
- **`out`** – the output channel. If *None*, one with no buffering will be created.
- **`close`** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

`take_while`(*p*, ***, *out=None*, *close=True*)
Returns a channel containing values *v* from the channel until *p(v)* becomes false.

Parameters

- **`p`** – a function receiving one element and returning whether this value should be kept.
- **`out`** – the output channel. If *None*, one with no buffering will be created.

- **close** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

drop_while (*p*, ***, *out=None*, *close=True*)

Returns a channel containing values *v* from the channel after *p(v)* becomes false for the first time.

Parameters

- **p** – a function receiving one element and returning whether this value should be dropped.
- **out** – the output channel. If *None*, one with no buffering will be created.
- **close** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

distinct (***, *out=None*, *close=True*)

Returns a channel containing distinct values from the channel (consecutive duplicates are dropped).

Parameters

- **out** – the output channel. If *None*, one with no buffering will be created.
- **close** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

reduce (*f*, *init=None*, ***, *out=None*, *close=True*)

Returns a channel containing the single value that is the reduce (i.e. left-fold) of the values in the channel.

Parameters

- **f** – a function taking two arguments *accumulator* and *next_value* and returning *new_accumulator*.
- **init** – if given, will be used as the initial accumulator. If not given, the first element in the channel will be used instead.
- **out** – the output channel. If *None*, one with no buffering will be created.
- **close** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

scan (*f*, *init=None*, ***, *out=None*, *close=True*)

Similar to *reduce*, but all intermediate accumulators are put onto the out channel in order as well.

Parameters

- **f** – a function taking two arguments *accumulator* and *next_value* and returning *new_accumulator*.
- **init** – if given, will be used as the initial accumulator. If not given, the first element in the channel will be used instead.
- **out** – the output channel. If *None*, one with no buffering will be created.
- **close** – whether *out* should be closed when there are no more values to be produced.

Returns the output channel.

dup ()

Create a `aiochan.channel.Dup()` from the channel

Returns the duplicator

pub(*topic_fn=operator.itemgetter(0)*, *buffer=None*, *buffer_size=None*)
Create a `aiochan.channel.Pub()` from the channel

Returns the publisher

distribute(**outs*, *close=True*)
Distribute the items in this channel to the output channels. Values will not be "lost" due to being put to closed channels.

Parameters

- **outs** – the output channels
- **close** – whether to close the output channels when the input closes

Returns self

collect(*n=None*)
Coroutine. Collect the elements in the channel into a list and return the list.

Parameters **n** – if given, will take at most *n* elements from the channel, otherwise take until channel is closed.

Returns an awaitable containing the collected values.

aiochan.channel.**tick_tock**(*seconds*, *start_at=None*, *loop=None*)
Returns a channel that gives out values every *seconds*.

The channel contains numbers from 1, counting how many ticks have been passed.

Parameters

- **start_at** – if *None*, the first tick occurs *seconds* later. If given, the first tick occurs at the given time (in float).
- **seconds** – time interval of the ticks
- **loop** – you can optionally specify the loop on which the returned channel is intended to be used.

Returns the tick channel

aiochan.channel.**timeout**(*seconds*, *loop=None*)
Returns a channel that closes itself after *seconds*.

Parameters

- **seconds** – time before the channel is closed
- **loop** – you can optionally specify the loop on which the returned channel is intended to be used.

Returns the timeout channel

aiochan.channel.**from_iter**(*it*, ***, *loop=None*)
Convert an iterable into a channel.

The channel will be closed on creation, but gets will succeed until the iterable is exhausted.

It is ok for the iterable to be unbounded.

Parameters

- **it** – the iterable to convert.
- **loop** – you can optionally specify the loop on which the returned channel is intended to be used.

Returns the converted channel.

`aiochan.channel.`**`from_range`**(*start=None*, *end=None*, *step=None*, *, *loop=None*)

returns a channel that gives out consecutive numerical values.

If *start* is *None*, then the count goes from *0* to the maximum number that python can count.

If *start* and *step* are given, then the values are produced as if by *itertools.count*.

Otherwise the values are produced as if by *range*.

Parameters **`loop`** – you can optionally specify the loop on which the returned channel is intended to be used.

Returns the range channel

`aiochan.channel.`**`select`**(**chan_ops*, *priority=False*, *default=None*, *cb=None*, *loop=None*)

Asynchronously completes at most one operation in chan_ops

Parameters

- **`chan_ops`** – operations, each is either a channel in which a get operation is attempted, or a tuple (chan, val) in which a put operation is attempted.
- **`priority`** – if True, the operations will be tried serially, else the order is random
- **`default`** – if not None, do not queue the operations if they cannot be completed immediately, instead return a future containing SelectResult(val=default, chan=None).
- **`cb`** –
- **`loop`** – asyncio loop to run on

Returns a function containing SelectResult(val=result, chan=succeeded_chan)

`aiochan.channel.`**`merge`**(**inputs*, *out=None*, *close=True*)

Merge the elements of the input channels into a single channel containing the individual values from the inputs.

Parameters

- **`inputs`** – the input channels
- **`out`** – the output chan. If *None*, a new unbuffered channel will be used.
- **`close`** – whether to close *out* when all inputs are closed.

Returns the ouput channel

`aiochan.channel.`**`zip_chans`**(**inputs*, *out=None*, *close=True*)

Merge the elements of the input channels into a single channel containing lists of individual values from the inputs. The input values are consumed in lockstep.

Parameters

- **`inputs`** – the input channels
- **`out`** – the output chan. If *None*, a new unbuffered channel will be used.
- **`close`** – whether to close *out* when all inputs are closed.

Returns the ouput channel

`aiochan.channel.`**`combine_latest`**(**inputs*, *out=None*, *close=True*)

Merge the elements of the input channels into a single channel containing lists of individual values from the inputs. The input values are consumed individually and each time a new value is consumed from any inputs, a list containing the latest values from all channels will be returned. In the list, channels that has not yet returned any values will have their corresponding values set to *None*.

Parameters

- **`inputs`** – the input channels
- **`out`** – the output chan. If *None*, a new unbuffered channel will be used.
- **`close`** – whether to close *out* when all inputs are closed.

Returns the ouput channel

class `aiochan.channel.`**`Dup`**(*inp*)

A duplicator: takes values from the input, and gives out the same value to all outputs.

Note that duplication is performed in lockstep: if any of the outputs blocks on put, the whole operation will block. Thus the outputs should use some buffering as appropriate for the situation.

When there are no output channels, values from the input channels are dropped.

Parameters **`inp`** – the input channel

`inp`

Returns the input channel

`tap`(*out=None*, *close=True*)

add channels to the duplicator to receive duplicated values from the input.

Parameters

- **`out`** – the channel to add. If *None*, an unbuffered channel will be created.
- **`close`** – whether to close the added channels when the input is closed

Returns the output channel

`untap`(*out*)

remove output channels from the duplicator so that they will no longer receive values from the input.

Parameters **`out`** – the channel to remove

Returns the removed channel

`untap_all`()

remove all output channels from the duplicator.

Returns *self*

`close`()

Close the duplicator.

Returns *self*

class `aiochan.channel.`**`Pub`**(*inp*, ***, *topic_fn=operator.itemgetter(0)*, *buffer=None*, *buffer_size=None*)

A publisher: similar to a duplicator but allowing for topic-based duplication.

As in the case of duplicators, the duplication process for any particular topic is processed in lockstep: i.e. if any particular subscriber blocks on put, the whole operation is blocked. Hence buffers should be used in appropriate situations, either globally by setting the *buffer* and *buffer_size* parameters, or individually for each subscription channel.

Parameters

- **`inp`** – the channel to be used as the source of the publication.
- **`topic_fn`** – a function accepting one argument and returning one result. This will be applied to each value as they come in from *inp*, and the results will be used as topics for

subscription. *None* topic is not allowed. If *topic_fn* is *None*, will assume the values from *inp* are tuples and the first element in each tuple is the topic.

- **`buffer`** – together with *buffer_size*, will be used to determine the buffering of each topic. The acceptable values are the same as for the constructor of `aiochan.channel.Chan()`.
- **`buffer_size`** – see above

`sub`(*topic*, *out=None*, *close=True*)
Subscribe *outs* to *topic*.

Parameters

- **`topic`** – the topic to subscribe
- **`out`** – the subscribing channel. If *None*, an unbuffered channel will be used.
- **`close`** – whether to close these channels when the input is closed

Returns the subscribing channel

`unsub`(*topic*, *out*)
Stop the subscription of *outs* to *topic*.

Parameters

- **`topic`** – the topic to unsubscribe from
- **`out`** – the channel to unsubscribe

Returns the unsubscribing channel

`unsub_all`(*topic*)
Stop all subscriptions under a topic

Parameters **`topic`** – the topic to stop. If *None*, all subscriptions are stopped.

Returns *self*

`close`()
close the subscription

Returns *self*

`aiochan.channel.`**`go`**(*coro*, *loop=None*)
Spawn a coroutine in the specified loop. The loop will stop when the coroutine exits.

Parameters

- **`coro`** – the coroutine to spawn.
- **`loop`** – the event loop to run the coroutine, or the current loop if *None*.

Returns An awaitable containing the result of the coroutine.

`aiochan.channel.`**`nop`**()
Useful for yielding control to the scheduler. :return:

`aiochan.channel.`**`run_in_thread`**(*coro*, *loop=None*)
Spawn a coroutine in the specified loop on a background thread. The loop will stop when the coroutine exits, and then the background thread will complete.

Parameters

- **`coro`** – the coroutine to spawn.
- **`loop`** – the event loop to run the coroutine, or a newly created loop if *None*.

Returns *(loop, thread)*, where *loop* is the loop on which the coroutine is run, *thread* is the thread on which the loop is run.

`aiochan.channel.`**`run`**`(`*coro*, *loop=None*`)`
Run coroutine in loop on the current thread. Will block until the coroutine is complete.

Parameters

- **`coro`** – the coroutine to run
- **`loop`** – the event loop to run the coroutine, or a newly created loop if *None*.

Returns *None*.

`aiochan.channel.`**`MAX_OP_QUEUE_SIZE = 1024`**
The maximum pending puts or pending takes for a channel.

Usually you should leave this option as it is. If you find yourself receiving exceptions due to put/get queue size exceeding limits, you should consider using appropriate *`aiochan.buffers`* when creating the channels.

`aiochan.channel.`**`MAX_DIRTY_SIZE = 256`**
The size of cancelled operations in put/get queues before a cleanup is triggered (an operation can only become cancelled due to the *`aiochan.channel.select()`* or operations using it, or in other words, there is no direct user control of cancellation).

4.2 Buffer

`class` `aiochan.buffers.`**`AbstractBuffer`**
Abstract buffer class intended for subclassing, to be used by channels.

`add``(`*el*`)`
Add an element to the buffer.

Will only be called after *can_add* returns *True*.

Parameters **`el`** – the element to add

Returns *None*

`take``()`
Take an element from the buffer.

Will only be called after *can_take* returns *True*. :return: an element from the buffer

`can_add`
Will be called each time before calling *add*.

Returns bool, whether an element can be added.

`can_take`
Will be called each time before calling *take*.

Returns bool, whether an element can be taken.

`class` `aiochan.buffers.`**`FixedLengthBuffer`**`(`*maxsize*`)`
A fixed length buffer that will block on get when empty and block on put when full.

Parameters **`maxsize`** – size of the buffer

`class` `aiochan.buffers.`**`DroppingBuffer`**`(`*maxsize*`)`
A dropping buffer that will block on get when empty and never blocks on put.

When the buffer is full, puts will succeed but the new values are dropped.

Parameters **`maxsize`** – size of the buffer

`class` `aiochan.buffers.`**`SlidingBuffer`**`(`*maxsize*`)`

A sliding buffer that will block on get when empty and never blocks on put.

When the buffer is full, puts will succeed and the oldest values are dropped.

Parameters **`maxsize`** – size of the buffer

`class` `aiochan.buffers.`**`PromiseBuffer`**`(`*_=None*`)`

A promise buffer that blocks on get when empty and never blocks on put.

After a single value is put into the buffer, *all* subsequent gets will succeed with this value, and *all* subsequent puts will succeed but new values are ignored.

`class` `aiochan.buffers.`**`IterBuffer`**`(`*it*`)`

A buffer that is constructed from a iterable (unbounded iterable is ok).

The buffer never accepts new inputs and will give out items from the iterable one by one, and when the iterable is exhausted will block on further gets.

Parameters **`it`** – the iterable to construct the buffer from.

Python Module Index

Index