A simple introduction to Python’s asyncio – Hacker Noon

Originalartikel

Backup

<html> <figure name=„6d7c“ id=„6d7c“ class=„graf graf–figure graf–leading“><div class=„aspectRatioPlaceholder is-locked c3“> </figure> <blockquote name=„c8f5“ id=„c8f5“ class=„graf graf–pullquote graf-after–h3“ readability=„4“> This is a no-buzzword first principles introduction to the asyncio library in Python. </blockquote> <p name=„9d6b“ id=„9d6b“ class=„graf graf–p graf-after–pullquote“>If you’ve come here, it is likely that you have heard of words such as asynchronous, concurrency and parallelism. Before we start off with asyncio, lets quickly get some basic things about these words right (via examples), so that we have a solid foundation to build this upon. <p name=„1083“ id=„1083“ class=„graf graf–p graf-after–p“><strong class=„markup–strong markup–p-strong“>Concurrency is like having two threads running on a single core CPU. Instructions from each thread could be interleaved, but at any given time, only one of the two threads is actively making progress. <p name=„b452“ id=„b452“ class=„graf graf–p graf-after–p“><strong class=„markup–strong markup–p-strong“>Parallelism is like having two threads running simultaneously on different cores of a multi-core CPU. <blockquote name=„8ce6“ id=„8ce6“ class=„graf graf–blockquote graf-after–p“ readability=„4“> It is important to note that parallelism implies concurrency but not the other way round. </blockquote> <p name=„dfae“ id=„dfae“ class=„graf graf–p graf-after–blockquote“><strong class=„markup–strong markup–p-strong“>Asynchronous is a higher level programming concept, where you fire off some task, and decide that while you don’t have the result of that task, you are better off doing some other work instead of waiting. <blockquote name=„1825“ id=„1825“ class=„graf graf–blockquote graf-after–p“ readability=„7“> When you do things asynchronously, you are, by definition implying concurrency between those things. </blockquote> <h4 name=„6e25“ id=„6e25“ class=„graf graf–h4 graf-after–blockquote“>Why asynchronous programming?</h4> <p name=„db7a“ id=„db7a“ class=„graf graf–p graf-after–h4“>Why do we want to write asynchronous programs you say — because it could increase the performance of your program many many times. Imagine you have a single core machine you are running your app on. You receive a request, and you need to make two database queries to fulfil that request. Each query takes 50ms of time. With a synchronous program, you would make the second request only after completing the first — total time 100ms. With an asynchronous program, you could fire off both the queries one after the other — total time 50ms. <h3 name=„1363“ id=„1363“ class=„graf graf–h3 graf-after–p“>asyncio</h3> <p name=„3b23“ id=„3b23“ class=„graf graf–p graf-after–h3“>Asyncio is all about writing asynchronous programs in Python. Asyncio is a beautiful symphony between an <strong class=„markup–strong markup–p-strong“>Event loop, <strong class=„markup–strong markup–p-strong“>Tasks and <strong class=„markup–strong markup–p-strong“>Coroutines all coming together so perfectly — its going to make you cry. <h4 name=„3546“ id=„3546“ class=„graf graf–h4 graf-after–p“>The Event Loop</h4> <p name=„0108“ id=„0108“ class=„graf graf–p graf-after–h4“>This is what makes it all possible — a simple loop, thats it. Well not <em class=„markup–em markup–p-em“>that simple. But here is how it works. The event loop is the orchestrator of the symphony. It runs <em class=„markup–em

markup–p-em“>tasks one after the other. At any given time, only one of the tasks is running. <p name=„2f82“ id=„2f82“ class=„graf graf–p graf-after–p“>As you can imagine, there is a lot of pressure on the active task, since other tasks are waiting for their turn. So, when the active task makes a blocking call, say a network request, and cannot make further progress it gives the control back to the event loop realising that some other task could possibly better utilise the event loop’s time. It also tells the event loop what exactly it is blocked upon, so that when the network response comes, the event loop can consider giving it time to run again. <blockquote name=„656f“ id=„656f“ class=„graf graf–blockquote graf-after–p“ readability=„8“> The event loop time is precious. If you are not making progress, you should step off the loop, so that someone else can. Event loop is the measure of progress. </blockquote> <h4 name=„b11e“ id=„b11e“ class=„graf graf–h4 graf-after–blockquote“>The Coroutine & Task</h4> <p name=„e086“ id=„e086“ class=„graf graf–p graf-after–h4“>Coroutines (co-operative routines) are a key element of the symphony. It is the coroutines, and their co-operative nature, that enables giving up control of the event loop, when the coroutine has nothing useful to do. A coroutine is a stateful generalisation of the concept of subroutine. <p name=„3689“ id=„3689“ class=„graf graf–p graf-after–p“>A subroutine is your good old-fashioned function or method. You invoke the subroutine to perform a computation. You may invoke it again, but it does not hold state between the two invocations. Every invocation is a fresh one and same computation is performed. <p name=„e1d3“ id=„e1d3“ class=„graf graf–p graf-after–p“>A coroutine, on the other hand, is a cute little <strong class=„markup–strong markup–p-strong“>stateful widget. It looks like a subroutine, but it maintains state in between executions. In other words, when a coroutine “returns” (yields control) it simply means that it has <strong class=„markup–strong markup–p-strong“>paused its execution (with some saved state). So when you “invoke” (give control to) the coroutine subsequently, it would be correct to say that the coroutine has <strong class=„markup–strong markup–p-strong“>resumed its execution (from the saved state). <blockquote name=„e254“ id=„e254“ class=„graf graf–blockquote graf-after–p“ readability=„6“> Coroutines look like a normal function, but in their behaviour they are stateful objects with

markup--blockquote-code"

```
resume()
```

and

markup--blockquote-code"

```
pause()
```

 — like methods. </blockquote> <p name=„439f“ id=„439f“ class=„graf graf–p graf-after–blockquote“>In Python 3.5+, the way a <strong class=„markup–strong markup–p-strong“>coroutine pauses itself is using the

markup--p-code"

```
await
```

keyword. Inside a coroutine, when you

markup--p-code"

```
await
```

on another coroutine, you step off the event loop and schedule the awaited coroutine to run <strong class=„markup–strong markup–p-strong“>immediately. That is, an

markup--p-code"

```
await other_coroutine
```

inside a coroutine will pause it, and schedule the coroutine

markup--p-code"

```
other_coroutine
```

to run immediately. <blockquote name=„0dc5“ id=„0dc5“ class=„graf graf–blockquote graf-after–p“ readability=„4“> Note that the event loop does not preempt a running coroutine. Only a coroutine can pause itself. </blockquote> <p name=„eb7e“ id=„eb7e“ class=„graf graf–p graf-after–blockquote“>Below is a very simple example <em class=„markup–em markup–p-em“>(Python 3.5+) of how coroutines cooperate with each other. We will use a pre-defined coroutine

markup--p-code"

```
asyncio.sleep
```

to help us simulate blocking tasks for this example, but it could be anything in a real world scenario like a network request, db query etc. <p name=„75c1“ id=„75c1“ class=„graf graf–p graf-after–p“>Note that the <strong class=„markup–strong markup–p-strong“>code runs in a single thread and yet, the output will have interleaved print statements. This happens because when a coroutine gets blocked, it steps off the loop, so that the other one can run (yay! asynchronous programming with asyncio). <figure name=„977e“ id=„977e“ class=„graf graf–figure graf–iframe graf-after–p“> <figcaption class=„imageCaption“>python run coroutine_example.py</figcaption></figure><p name=„72f7“ id=„72f7“ class=„graf graf–p graf-after–figure“>Some points to note <ul class=„postList“><li name=„05d5“ id=„05d5“ class=„graf graf–li graf-after–p“>Calling a coroutine definition <strong class=„markup–strong markup–li-strong“>does not execute it. It initialises a <em class=„markup–em markup–li-em“>coroutine object. You

markup--li-code"

```
await
```

on <em class=„markup–em markup–li-em“>coroutine objects, not <em class=„markup–em markup–li-em“>coroutine definition as you can see in

markup--li-code"

```
line 8
```

and

markup--li-code"

```
line 17
```

above. <li name=„e2e9“ id=„e2e9“ class=„graf graf–li graf-after–li“><strong class=„markup–strong markup–li-strong“>Event loop runs tasks, not <em class=„markup–em markup–li-em“>coroutine objects directly. Tasks are a wrapper around <em class=„markup–em markup–li-em“>coroutine objects. When you write

markup--li-code"

```
await coroutine_object
```

you essentially schedule a wrapper task to be run on the event loop <strong class=„markup–strong markup–li-strong“>immediately. <li name=„e1e7“ id=„e1e7“ class=„graf graf–li graf-after–li“>

markup--li-code"

```
asyncio.sleep
```

is a coroutine as well, provided by the asyncio library.

markup--li-code"

```
asyncio.sleep(2)
```

initialises a coroutine object with a value of 2 seconds. When you

markup--li-code"

```
await
```

on it, you give control of the event loop to it. Sleep coroutine is smart and does not block the loop. It immediately releases control, simply asking the loop to wake it up after the specified time. When the time expires, it is given back the control and it immediately returns, thereby unblocking its caller (in the above example

markup--li-code"

```
coroutine_1
```

or the

markup--li-code"

```
coroutine_2
```

). <li name=„f140“ id=„f140“ class=„graf graf–li graf-after–li“>The above example had three different types of coroutines that ran on the event loop —

markup--li-code"

```
coroutine_1
```

,

markup--li-code"

```
coroutine_2
```

and

markup--li-code"

```
asyncio.sleep
```

. However, four different tasks ran on the loop, corresponding to the following coroutine objects —

markup--li-code"

```
coroutine_1()
```

and

markup--li-code"

```
coroutine_2()
```

scheduled at

markup--li-code"

```
line 25
```

,

markup--li-code"

```
asyncio.sleep(4)
```

scheduled at

markup--li-code"

```
line 8
```

and

markup--li-code"

```
asyncio.sleep(5)
```

scheduled at

markup--li-code"

```
line 17
```

. <li name=„a37f“ id=„a37f“ class=„graf graf–li graf-after–li“>Another way to schedule tasks (though not immediately) on the loop is using the

markup--li-code"

```
<a
href="https://docs.python.org/3/library/asyncio-task.html#asyncio.ensur
e_future" data-
href="https://docs.python.org/3/library/asyncio-task.html#asyncio.ensur
```

```
e_future" class="markup--anchor markup--li-anchor" rel="noopener"
target="_blank">ensure_future()</a>
```

or the

markup--li-code"

```
<a
href="https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.
AbstractEventLoop.create_task" data-
href="https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.
AbstractEventLoop.create_task" class="markup--anchor markup--li-anchor"
rel="noopener" target="_blank">AbstractEventLoop.create_task()</a>
```

methods, both of which accept a coroutine object. Example code in the end demonstrates these methods. <h4 name=„f57b“ id=„f57b“ class=„graf graf–h4 graf-after–li“>A more realistic yet simple example</h4> <figure name=„49df“ id=„49df“ class=„graf graf–figure graf–iframe graf-after–h4“> <figcaption class=„imageCaption“>python run asyncio_example.py</figcaption></figure><h4 name=„6661“ id=„6661“ class=„graf graf–h4 graf-after–figure“>Python at ArchSaber</h4> <p name=„6005“ id=„6005“ class=„graf graf–p graf-after–h4“>At ArchSaber one of our aim has always been to dig insights deep from the application code of our customers. A lot of our clients depend upon our APM solution for Python. As a result we make great efforts in understanding the intricacies of the language and the frameworks around it. We ourselves rely heavily on Python — a lot of our analytics engine and ML code is written in Python, through which we push real-time root cause analysis to our clients’ production issues. <blockquote name=„470c“ id=„470c“ class=„graf graf–pullquote graf-after–p“ readability=„5“> Thanks for reading. If you like this post, please subscribe and share. </blockquote> <figure name=„09ef“ id=„09ef“ class=„graf graf–figure graf–iframe graf-after–pullquote“> </figure><figure name=„85da“ id=„85da“ class=„graf graf–figure graf-after–figure graf–trailing“><div class=„aspectRatioPlaceholder is-locked c6“> </figure> </html>