PyMiami

David Gutierrez

Sep 06, 2021

TABLE OF CONTENTS

CHAPTER ONE

SECOND PART: PYTHON ASYNC I/O

Created by David Gutierrez for PyMiami.
PyMiami has been holding monthly FREE events about Python programming language since 2018.

Sponsors:

- FIU School of Engineering and Computers
- Python Software Solutions

1.1 PREFACE

1.2 Contact Information

Author: David Gutierrez
email: david @ pymiami . org

Join our PyMiami Python Developers group's social accounts to stay updated about upcoming events:

Twitter

Facebook

LinkedIn

Do you need help creating your Python applications ?

Do you need training?

Contact me at david @ pythonsoftware.solutions

1.3 Important Note

All content in this tutorial has been taken from the references named in the "References" section. What I have done is researched for the best, most relevant information and organized it in a way that better helps understand the subject.

I find this to be more efficient than to re-create the same content. In addition, since the creators of these references are the best of the best, you will be receiving unparalleled information which is prominent to what I could provide.

This course is also a way of recognizing the creators of the content and their life-long contribution to Python. Please take a moment to follow their social media, links below,and check out heir companies, courses, trainings and what they do. You will benefit immensely.

I will start this article by mentioning all the references and documentation used in this Tutorial.

1.4 References

- 0- Guido, creator of everything.
- 1- Yury Selivanov, Async await and asyncio in Python 3.6 and beyond PyCon 2017.
- 2- Miguel Grinberg, Asynchronous Python for the Complete Beginner PyCon 2017.
- 3- John Reese, Thinking Outside the GIL with AsyncIO and Multiprocessing - PyCon 2018.
- 4- Mariatta Wijaya, Hands-on Intro to aiohttp - PyCon 2019] video.
- 5- Andrew Svetlov, Hands-on Intro to aiohttp - PyCon 2019] slides.
- 6- Luciano Ramalho, Fluent Python 1st Edition
- 7- Brett Cannon, How async/await works in Python3.5
- 8- David Beazley, Keynote at PyCon Brazil 2015
- 9- David Beazley, Curios Course on Coroutines and Concurrency
- 10- Łukasz Langa, Thinking In Coroutines - PyCon 2016
- 11- Brett Cannon, How the heck does async/await work in Python 3.5?
- 12- Raymond Hettinger, Keynote on Concurrency, PyBay 2017
- 13 -Doug Hellman, The Python 3 Standard Library by Example
- 14 -Dusty Phillips, Python 3 Object-Oriented Programming 3rd Edition.
- 15 -Kenneth Reitz, Kenneth Reitz's requests library

1.5 Documents and Papers

- 100- certifi documentation.
- 101- SSL context documentation.
- 102- aiohttp documentation.
- 103- PEP 492, Coroutines with async and await syntax,.
- 104- Python Event loop.
- 105- Coroutines and tasks.

- 106- PEP 380, Syntax for Delegating to a Sub-generator.

1.6 A quick review

In the the first part of this talk we'll review the origins of async I/O in Python.

We'll see how Python first achieved multiprocessing capabilities with threads , and then with async I/O

We'll see how the evolution of iterations concepts (iterables, iterator and generators functions), created the path for coroutines.

All that was missing was an event loop , and all of the sudden Python had asyncio. A master strategy, that ended with the dismiss of generators, deterred and renamed from the asynchronous world.

1.7 At the beginning there were threads..

In our first talk about this subject, "Talk #1 - Concurrency with Threads and Futures, Feb 8th, 2020.", we discussed the evolution of concurrency in Python.

We started by analyzing Chapter 17, and the first part of Chapter 18 of Luciano Ramalho's book "Fluent Python" [6]. In Chapter 17 "Concurrency and Features", Luciano starts by explaining a sequential download. It is, of course, necessary that we see a sequential example and how we could later improve it using concurrency.

Luciano starts Chapter 18 with an example of Threads, and later move us to coroutines and finally async I/O. We discussed Chapter 17 and used the treads example in Chapter 18 to learn about Threads and Futures. We discussed Raymond Hettinger [12] talk, "Keynote on Concurrency, PyBay 2017", where we learned the complexity and perils that lure on the Threads world.

Finally, we were able to arrive to below evolutionary conclusion in Table 1. If you want to know more about this, we encourage you to read our talk and the resources mentioned.

Table 1: Table 1. Concurrency in Python: Threads , Futures and Async I/O

Python	Python 3.2/ 2011	Python 3.4/2014 Created by Guido
	current.futures ThreadpoolExecutor Context Manager	asyncio
Threads, Locks,Semaphore,Event,Timer,Queue	Threads, Locks,Semaphore,Event,Timer,Queue	Coroutines, AsyncIO not blocking loops, call backs
Multi Thread Processing	Multi Thread Processing	Single Thread Processing

We are here, one month later , to talk about async I/O, far away from the thread and blocked world we discussed in February, entering into a new universe of cooperation multitasking and event loops.

But then, we wonder, what is Async I/O and what does Python have to do with it?

1.8 . . . And then we were given a breath of life Async I/O

What is async I/O ?

According to Wikipedia, “In computer science, asynchronous I/O (also non-sequential I/O) is a form of input/output processing that permits other processing to continue before the transmission has finished”.

David Beazley, “Python Brasil 2015 keynote. . . ” [8] , starts by recounting that Async I/O was implemented by C# a long time ago . Łukasz Langa, “Thinking In Coroutines - PyCon 2016” [6] states that those who knows JavaScript/ NodeJS async I/O should easily get the concept in Python.

Async I/O is not a Python concept and indeed has been implemented in other languages a long time ago.

Async I/O gives us the possibility of writing concurrent programs without using Threads. Threads belong to the OS and ,as such, are more expensive and less scalable than async I/O cooperative multitasking . So just using one thread , the main thread, we are able to gain multiprocessing capability.

Before going any step further ,take a moment to look at Miguel Grinberg’s, Asynchronous Python for the Complete Beginner talk [2] where, in just a few minutes, Miguel quickly describes the universe of async programing.

But then we want the details, the “how was this possible?”, and the desire to understand how async I/O works ?

1.9 How the heck does async/await work in Python ? Brett Cannon [11]

When you feel afraid of not knowing enough about async I/O , it will be a relief to know that Brett Cannon asked himself the above question in [11]. Once you know who he is, you will feel an immediate peace of mind on having any doubts about async I/O. He too had doubts at some point!

David Beazley in [8] and [9], clearly describes the evolution of async I/O. Meanwhile, Brett Cannon in [11] starts by taking us back to the origin and uses of ‘yield from’ in coroutines. David Beazley is the one that takes us even further back by starting for a differentiation between generators and coroutines.

David Beazley makes the distinction that ‘generators’ and ‘generator function’ should be differentiated from coroutines in both its intention and uses. We will see this distinction below.

1.10 Iterables and Iterator and Generator Function

Generators are intrinsically linked to iteration. And that is the reason we start by analyzing the meaning of iterable and iterator in Python.

Luciano Ramalho in [6] , Chapter 14 clearly defined iterable and iterator concepts. But before going there take note the name of that Chapter :

“Iterables, Iterator and Generators”.

iterable:

Any object from which the iter built-in function can obtain an iterator. Objects implementing an __iter__ method returning an iterator are iterable. Sequences are always iterable; as are objects implementing a __getitem__ method that takes 0-based indexes.

It’s important to be clear about the relationship between iterables and iterators: Python obtains iterators from iterables.

In page 418 [6], Luciano stops to explain why Sequences are iterable and what the interpreter does when it needs to iterate over an object

Then in the same Chapter in page 423 he gives us a definition of iterator

iterator:

Any object that implements the __next__ no-argument method that returns the next item in a series or raises StopIteration when there are no more items. Python iterators also implement the __iter__ method so they are iterable as well.

Now,let's see an example and take note that the keyword "yield" has not been used for anything so far. Remember the title of the Chapter does not mention coroutines : "Iterables, Iterator and Generators".

1.11 Generator Functions (yield appears)

Python provides generator functions as a convenient shortcut to building iterators. For this Python uses yield

Simple generator function "countdown.py" , taken from David Beazley… [9]

```
#countdown.py
# A simple generator function

def countdown(n):
    print("Counting down from", n)
    while n > 0:
        yield n
        n -= 1
    print "Done counting down"

# Example use
if __name__ == '__main__':
    for i in countdown(10):
        print(i)
```

Note: Note the use of the yield to the left of the value. The yield statement is pausing execution of the countdown() function and returning back to the caller both the n value and the control. It is important to see that the state of the countdown function remains the same therefore when the caller returns control to the generator , the function continues execution after the yield n statement !!

A Python function that has the yield keyword in its body is a generator function. When this function is called it returns a generator object. In other words, a generator function is a generator factory, (…Luciano pag 428 [6])

But . . . .

people knew that if we took the "pausing" part of generators and added in a "send stuff back in" aspect to them, Python would suddenly have the concept of coroutines ..Brett [11]

When a generator yields , it keep its state.

As we have seen so far, generators are able to send values back to the caller. We'll see very soon that they will be able to accept values from the caller as well. But that will come with a price: they will be renamed to something called coroutines and deterred from the new world of async operations forever.

wanted to immigrate to Spanish colonies, had to change their names and last names from Robert Baker to Roberto Bequer. What an horror history is !!

1.12 Coroutines and its importance in async operations

- The origins of the yield from keyword

  In [6] , Chapter 16 Page 480, Luciano Ramalho writes "How Coroutines evolved form Generators" from that Lecture we learn that:

  ..In addition to .send(..), PEP 340 also added throw(..) and .close() methods that allowed a caller to send a datum into the generator, thrown an exception to be handled inside the generator, and terminate it.

  PEP 380 , makes two syntax changes to generators functions to make them more useful coroutines:

  - "yield from" syntax enables complex generators to be refactored into smaller nested generators .
  - A generator can now return a value. (previously attempts to return a value would raise a SyntaxError exception) See "return Result (count , average)" in the example below

  We can go deeper on the details of PEP 380, created by Gregory Ewing reading [16]

  - Coroutines are just special generators . . . . You send values into
  - Coroutines are very similar to generators in Python: they both use yield.
  - **Just the meaning of yield implies to 'let things go', to stop and let others continue. So, the yield keyword** has an asynchronous meaning. Brett [8] .
  - **Coroutines suspend execution in the yield statement and pass control to the caller** along with any value to the right of yield.

  - Normal coroutines in python use yield to wait to receive an elements from the caller. Usually yield is to the right
  - The caller send data to the coroutine using send.
  - Coroutines are functions whose execution you can pause.

  We could see some examples of use of coroutines now , but what will be really useful is to see David Beazley examples in "A Curious Course on Coroutines" [9] . . . , Part 1 : Introduction to Generators and Coroutines

  We'll see in this talk how we start separating coroutines from generators in its use and purpose, and how the former "evolved" to create async/await syntax to be used in async I/O operation in Python

  Then looking at Chapter [2] on David Bealey [9] , and the code in "copipe.py" we will see how how to hook up a pipeline with coroutines.

Note: Note that stacking coroutines , meaning having one coroutine doing something passing the result of its operation to another coroutine so it does another things, with that kind of structure , we could get pretty powerful functionality. "yield from" will allow us to successfully concatenate coroutines and get messages passed back and forth from the caller to any coroutine in the pipe.

Now is time we can understand and see "yield from" in Action , by taken a look to [6] Example 16-17, "Fluent Python", Luciano Ramalho Page [496]

```
def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None :
            break
        total + = term
        count + = 1
        average = total / count
        return Result ( count , average )

 # the delegating generator
 def grouper ( results , key ):
    while True :
        results [ key ] = yield from averager ( )

 # the client code, a.k.a. the caller

 def main ( data ):
    results = { }
    for key , values in data . items ( ):
    group = grouper ( results , key )
    next ( group )
    for value in values:
        group.send(value)
    group.send(None) # important!

    # print(results) # uncomment to debug
    report ( results )

    # output report
    def report ( results ):
        for key , result in sorted ( results . items ( ) ):
            group , unit = key . split ( ' ; ' )
            print ( ' {:2} {:5} averaging {:.2f}{} '.format(
                      result.count, group, result.average, unit))

 data = { ' girls;kg ':
                [ 40.9 , 38.5 , 44.3 , 42.2 , 45.2 , 41.7 , 44.5 , 38.0 , 40.
↪6 , 44.5 ] ,
          ' girls;m ':
                [ 1.6 , 1.51 , 1.4 , 1.3 , 1.41 , 1.39 , 1.33 , 1.46 , 1.45 ,
↪ 1.43 ] ,
           ' boys;kg ':
                 [ 39.0 , 40.8 , 43.2 , 40.8 , 43.1 , 38.6 , 41.4 , 40.6 ,␣
↪36.3 ] ,
           ' boys;m ':
                 [ 1.38 , 1.5 , 1.32 , 1.25 , 1.37 , 1.48 , 1.25 , 1.49 , 1.
↪46 ] , }
```

(continues on next page)

(continued from previous page)

```
if __name__ == ' __main__ ':
    main(data)
```

We can easily see now the power of "yield from" and how this keywords will allow us to extract values and send values directly to the subgenerator, which yield data back at the caller.

1.13 Interlude : Generators are not Coroutines

David Beazley, minute 30:42 of Curious Course on Coroutines and Concurrency [9]

"Coroutines evolved from generators.From coroutines a new syntax gave birth to async and await keywords and that is the reason we have taken a look to all that here so far."

But as David Beazley quoted in above references [9], generators are not coroutines.

Below Table shows the differences.

Table 2: Table 2 Differences between GENERATORS and COROUTINES

GENERATORS	COROUTINES
Produce data	Consume data
Related to iteration	Nothing to do with iteration,even when producing values and Related to async behavior

Python's coroutines are special functions that give up control to the caller without losing their state, Dough Hellman [13]

They allow to return information from a pipe of multiple coroutines using "yield from"

They consume data and evolved from generators

1.14 Getting started with Python async I/O

We are ready to dive into the world of async i/O in Python. So lets see how it really works and code some examples.

1.15 Event Loops

Doug Hellman in his mater book, "The Python 3 Standard Library by Example" (Developer's Library) 1st Edition, Chapter 10.5, describes what an Event Loop is:

"Event loop, a first-class object that is responsible for efficiently handling I/O events, system events, and application context changes. Several loop implementations are provided, to take advantage of the operating systems' capabilities efficiently. While a reasonable default is usually selected automatically, it is also possible to pick a particular event loop implementation from within the application". [13]

Python documentation describes the event loop as follow: “Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses”. [14]

“In computer science, the event loop is a programming construct or design pattern that waits for and dispatches events or messages in a program. The event loop works by making a request to some internal or external “event provider” (that generally blocks the request until an event has arrived), then calls the relevant event handler (“dispatches the event”).

The event loop is also sometimes referred to as the message dispatcher, message loop, message pump, or run loop.”

MDN gives us the following definition of JavaScript event loop:

“JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. “

In any case an event loop is as its name says , is a loop :-) . In Python we can implement loops with something so simple as a while statement . Also we can do it using a context Manager, etc.

In brief the Event Loop is an important part of any Async I/O architecture, and in Python this is also the case.

Specifically the asyncio module , one of many async i/o implementations in Python , creates the event loop in the BaseDefaultEventLoopPolicy class . This event loop runs in its own thread and what we expect from it (efficiency, simplicity and responsibility in task delegation) we certainly get it.

See below some lines of the source code for the method get_event_loop() in the BaseDefaultEventLoopPolicy class .

BaseDefaultEventLoop.

```python
class BaseDefaultEventLoopPolicy(AbstractEventLoopPolicy):
    """Default policy implementation for accessing the event loop.
    In this policy, each thread has its own event loop.  However, we
    only automatically create an event loop by default for the main
    thread; other threads by default have no event loop.
    Other policies may have different rules (e.g. a single global
    event loop, or automatically creating an event loop per thread, or
    using some other notion of context to which an event loop is
    associated).
    """

    _loop_factory = None

    class _Local(threading.local):
        _loop = None
        _set_called = False

    def __init__(self):
        self._local = self._Local()

    def get_event_loop(self):
        """Get the event loop for the current context.
        Returns an instance of EventLoop or raises an exception.
        """
        if (self._local._loop is None and
                not self._local._set_called and
                isinstance(threading.current_thread(), threading._MainThread)):
            self.set_event_loop(self.new_event_loop())

        if self._local._loop is None:
            raise RuntimeError('There is no current event loop in thread %r.'
                               % threading.current_thread().name)
```

(continues on next page)

(continued from previous page)

```
        return self._local._loop
```

David Beazley curio library uses an event loop object he called the Kernel. We ask the Kernel to run and pass as a parameter to it, a coroutine. One more time the Kernel (Event Loop) has only limited coordination and functionality and is build on top of simple data structures, threads.Event object (real threads) etc

The curio Event Loop (Kernel) run function

```
def run(corofunc, *args, with_monitor=False, selector=None,
        debug=None, activations=None, **kernel_extra):
    '''
    Run the curio kernel with an initial task and execute until all
    tasks terminate.  Returns the task's final result (if any). This
    is a convenience function that should primarily be used for
    launching the top-level task of a curio-based application.  It
    creates an entirely new kernel, runs the given task to completion,
    and concludes by shutting down the kernel, releasing all resources used.

    Don't use this function if you're repeatedly launching a lot of
    new tasks to run in curio. Instead, create a Kernel instance and
    use its run() method instead.
    '''
    kernel = Kernel(selector=selector, debug=debug, activations=activations,
                    **kernel_extra)

    # Check if a monitor has been requested
    if with_monitor or 'CURIOMONITOR' in os.environ:
        from .monitor import Monitor
        m = Monitor(kernel)
        kernel._call_at_shutdown(m.close)
        kernel.run(m.start)

    with kernel:
        return kernel.run(corofunc, *args)
```

We have mentioned 3 Event loops two in Python and one in javascript the asyncio event loop is able to run asynchronous code (threads , futures) as well as async i/o not blocking code

This is taken from the Python documentation

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
```

(continues on next page)

(continued from previous page)

```
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

As we can see, we used the asyncio Event Loop to run Process and Threads!!

Let's see some more examples of Even Loops.

From Python asyncio docs :

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

More examples:

1.16 His name is Yuri Selivanov, or, "async / await to take us outside of the chaos"

PEP 492 [13] was introduced by Yuri Selivanov, to save us, the rest of the mortals, from the confusions and darkness. You can see also Yuri Selivanov in our first reference here. Go and take a look at Python PEP and search async and its authors. You will find his name close to multiples PEP

In the abstract we can read

" . . . [PEP 492] It is proposed to make coroutines a proper standalone concept in Python, and introduce new supporting syntax. The ultimate goal is to help establish a common, easily approachable, mental model of asynchronous programming in Python and make it as close to synchronous programming as possible.

This PEP assumes that the asynchronous tasks are scheduled and coordinated by an Event Loop similar to that of stdlib module asyncio.events.AbstractEventLoop. While the PEP is not tied to any specific Event Loop implementation, it is relevant only to the kind of coroutine that uses yield as a signal to the scheduler, indicating that the coroutine will be waiting until an event (such as IO) is completed."

In brief : async await is a new syntax created in Python to work with coroutines, just to make it clear it is a coroutine (and as we already know reinforce the idea of asymmetric behavior) , and not a generator.

Dusty Phillips, in [14] (p. 295) note:

There is an alternate syntax for coroutines using the async and await keywords. The syntax makes it clearer that the code is a coroutine and further breaks the deceiving symmetry between coroutines and generators. The syntax doesn't work very well without building a full event loop . . .

So in the new world of async, and in the newest version of Python , coroutines reinforce the sense of asynchronous behavior by using:

- async –> instead of –> "yield"
- await –> instead of –> "yield from" (This transfer control back to the caller, in asyncio world ,the Event Loop)
- "async def" is now used to declare a native coroutine
- "async def" functions are always coroutines, even without await expressions
- coroutine object is the object returned when we call a coroutine(like generator objects are returned from generators)

```python
import asyncio

async def echo_server():
    print('Serving on localhost:8000')
    await asyncio.start_server(handle_connection,
                               'localhost', 8000)

async def handle_connection(reader, writer):
    print('New connection...')

    while True:
        data = await reader.read(8192)

        if not data:
            break

        print('Sending {:.10}... back'.format(repr(data)))
        writer.write(data)
```

(continues on next page)

(continued from previous page)

```
loop = asyncio.get_event_loop()
loop.run_until_complete(echo_server())
try:
    loop.run_forever()
finally:
    loop.close()
```

Above is the example given in the PEP 492.

Now let's see and discuss other examples. Luciano in [6] , Exercise 18.2, created the same script using asyncio , and curio . (It is in the GitHub project of the book not in the book itself)

```
#!/usr/bin/env python3

# spinner_curio.py

# credits: Example by Luciano Ramalho inspired by
# Michele Simionato's multiprocessing example in the python-list:
# https://mail.python.org/pipermail/python-list/2009-February/538048.html
import curio

import itertools
import sys


async def spin(msg):  # <1>
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            await curio.sleep(.1)  # <2>
        except curio.CancelledError:  # <3>
            break
    write(' ' * len(status) + '\x08' * len(status))


async def slow_function():  # <4>
    # pretend waiting a long time for I/O
    await curio.sleep(3)  # <5>
    return 42


async def supervisor():  # <6>
    spinner = await curio.spawn(spin('thinking!'))  # <7>
    print('spinner object:\n ', repr(spinner))  # <8>
    result = await slow_function()  # <9>
    await spinner.cancel()  # <10>
    return result


def main():
    result = curio.run(supervisor)  # <12>
    print('Answer:', result)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

and now the same example using asyncio with “await” and the new coroutine definitions “async def”

```
#!/usr/bin/env python3

# spinner_await.py

# credits: Example by Luciano Ramalho inspired by
# Michele Simionato's multiprocessing example in the python-list:
# https://mail.python.org/pipermail/python-list/2009-February/538048.html

import asyncio
import itertools
import sys


async def spin(msg):  # <1>
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            await asyncio.sleep(.1)  # <2>
        except asyncio.CancelledError:  # <3>
            break
    write(' ' * len(status) + '\x08' * len(status))


async def slow_function():  # <4>
    # pretend waiting a long time for I/O
    await asyncio.sleep(3)  # <5>
    return 42


async def supervisor():  # <6>
    spinner = asyncio.ensure_future(spin('thinking!'))  # <7>
    print('spinner object:', spinner)  # <8>
    result = await slow_function()  # <9>
    spinner.cancel()  # <10>
    return result


def main():
    loop = asyncio.get_event_loop()  # <11>
    result = loop.run_until_complete(supervisor())  # <12>
    loop.close()
    print('Answer:', result)


if __name__ == '__main__':
    main()
```

We can see that both libraries make use of async / await syntax and each one implement the same functionality , an Async I/O execution the former using curio the latter using asyncio

1.17 So at the end, what we need to know so far

Before trying to understand anything else we need to go to David Beazley's curio GitHub implementation here. No worries , we are not going to try to understand it. But we will read something really interesting :

> "Curio - There Are Many Async Libraries, But This One is Mine"

Unfortunately on Thursday 20th, 2020 at around 7:00pm ET, just a few days after I had writen this part of the conference, David Beazley changed the message of curio to Curio - "It's the Sauce!"

Now you need to go to Brett Cannon essay in [7]. and ,read this twice:

[7]. . . .In that talk, David pointed out that async/await is really an API for asynchronous programming (which he reiterated to me on Twitter). What David means by this is that people shouldn't think that async/await as synonymous with asyncio, but instead think that asyncio is a framework that can utilize the async/await API for asynchronous programming.

David Beazley actually believes this idea of async/await being an asynchronous programming API that he has created the curio project to implement his own event loop. This has helped make it clear to me that async/await allows Python to provide the building blocks for asynchronous programming, but without tieing you to a specific event loop or other low-level details (unlike other programming languages which integrate the event loop into the language directly). This allows for projects like curio to not only operate differently at a lower level (e.g., asyncio uses future objects as the API for talking to its event loop while curio uses tuples), but to also have different focuses and performance characteristics (e.g., asyncio has an entire framework for implementing transport and protocol layers which makes it extensible while curio is simpler and expects the user to worry about that kind of thing but also allows it to run faster)."

So at this point we should be clear of the following:

- What is Async I/O ? (read above we discussed this concept before)
- What is an Event Loop ? (read above we discussed this concept before)
- async/await is an asynchronous programming API
- async/await allows Python to provide the building blocks for asynchronous programming
- asyncio is a framework that can utilize the async/await API for asynchronous programming.
- curio is a library that uses the async/await API for asynchronous programming.
- trio is a library that uses the async/await API for asynchronous programming.
- aiohttp is an asynchronous HTTP client/server framework for asyncio

1.18 Python growing async I/O ecosystem

Here below we show some of the most used async I/O libraries and Python modules.

- Coroutines with async and await syntax
- Asynchronous Generators
- Asynchronous Comprehensions
- Asynchronous IO Support Rebooted: the "asyncio" Module
- Asynchronous I/O For subprocess.Popen

- third party libraries
- twisted
- gevent
- curio
- trio
- aiohttp
- aiosmtplib

1.19 Andrew Svetlov: aiohttp

In order to use asyncio, we must use libraries that are not blocking It means that those libraries or frameworks, need in turn to use asyncio's implementation of all I/O functions !!! So let say we want to use asyncio to get a web site. It should be easy with what we have done so far, right ? or not?

```
# Example similar to the Mariatta and Andrew exercises in [5]
import requests
import asyncio

async def fetch(pep):
    url = f"https://www.python.org/dev/peps/pep-{pep}/"
    await requests.get(url)


def main():
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(fetch(8010))
    loop.close()
    print('Answer:', result.content)

if __name__ == '__main__':
    main()
```

```
# OUTPUT
>>File "../asyncio/intro_aiohttp_handson_mariatta_pycon2019/3.1_wrong_asyncio_
↪requests.py", line 7, in fetch
>>await requests.get(url)
>>TypeError: object Response can't be used in 'await' expression
```

So what happened here ?

The reason for this is that requests is not awaitable .Requests comes from the another universe, the universe of blocking and uses sockets() that is blocking by nature, and as such we can not use it. So then what we are going to do.?

We have two options:

- We use a function or library that is awaitable (requests is not awaitable)
- We can call this function in another Thread or Process , so it will be immediately awaitable

So let's try to solve the issue using the second technique :

```
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor

_executor = ThreadPoolExecutor(1)


def fetch(pep):
    url = f"https://www.python.org/dev/peps/pep-{pep}/"
    return requests.get(url)


async def call_asyn():
    pep = 8010
    res = await loop.run_in_executor(_executor, fetch, pep)
    print(f"Finished downloading pep {pep}")
    return res


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(call_asyn())
    print('Answer:', result)
    loop.close()
```

I can see the stupor in your face.

Did not you say you were to take me to a new universe away from Threads and Process , and OS limitations to an unblocked world of cooperative multitasking ? Did not you show me what the Gods and Prophets, listed in that immense Reference above have said ? How is that you are talking about Threads and asyncio all mixed up together ?

Our friend requests (for humans) can not be used any longer in this new universe without pain. And meanwhile we await for a new and awaitable requests library for humans, from this artist that gracefully created it for us , Kenneth Reitz we have at least for now aiohttp.

Fortunately Andrew Svetlov created aiohttp and we can use it for cases like this. Andrew Svetlov and Mariatta Wijaya, in [4] and [5], provided us with some examples we will analyze later

But for now let's try to simply solve this problem .All that we want is to use asyncio to go and grap a response from a web site , but without using Threads , or Future or any other type of concurrency in Python.

```
import asyncio
import aiohttp


async def fetch():
    url = f"http://httpbin.org/get"
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            print(resp.status)
            print(await resp.text())


if __name__ == '__main__':
    asyncio.run(fetch())
```

And finally we get our first correct response using asyncio

Oh , but wait , it is not the same url neither the same requests, so tehre is no wait I can compare !!

Sorry about that , it will be easy to write our first example similar to the one we saw earlier Here we go

```python
import asyncio
import aiohttp


async def fetch(pep):
    url = f"https://www.python.org/dev/peps/pep-{pep}/"
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            print(resp.status)
            print(await resp.text())


if __name__ == '__main__':
    asyncio.run(fetch(8010))
```

Ah it did not work !! We are getting an TLS cetificate error

Well aiohttp is not linked to any specific certificate manager at the client side , so it does not kow how to validate the https.

In our friend requests library Kenneth Reitz, incorpotated the certificate validation for us using certifi so signatures of certificates could be verified using the installed root certificates source in our system.

It is simple fix in aiohttp

```python
import asyncio

import aiohttp
import ssl

ssl_ctx = ssl.SSLContext()

async def fetch(pep):
    url = f"https://www.python.org/dev/peps/pep-{pep}/"
    async with aiohttp.ClientSession() as session:
        async with session.get(url, ssl=ssl_ctx) as resp:
            print(resp.status)
            print(await resp.text())


if __name__ == '__main__':
    asyncio.run(fetch(8010))
```

Is it the end !! Well almost at this point you have the basic blocks to use this amazing library in Python for any multiprocessing.

And what you should do now?

- Every day new Python libraries, that support asyncio, are created, so go there and study them and use them.
- Go to Twitter and follow the amazing People that created this tool for us
- Buy the books I mentioned in this serie , they will really help you
- Check the authors talks in conferences
- And finally study deeper the resources provided

1.20 FAREWALL

Thank you for reading and I hope this has been useful.

If you think this document could be improved some way, please email me at david @ pymiami . org

Join our PyMiami Python Developers group's social accounts to stay updated about upcoming events:

> Twitter PyMiami
>
> LinkedIn PyMiami
>
> Facebook PyMiami
>
> LinkedIn Python Software Solutions

Do you need help creating your Python applications?
Do you need training?

Contact me at david @ pythonsoftware.solutions

Finally, join the social accounts of the authors named in the references.

Thanks !!
David Gutierrez

1.21 Indices and tables