**Weird Python Coroutines**

Let’s forget about asyncio . What is coroutine? How does coroutine work? First of all — coroutines are not asynchronous. Coroutines are not concurrent until they are. **Coroutines are suspendable,**and the possibility to suspend and resume computation allows them to be effectively used at concurrent and finally asynchronous programs.

How is it, suspendable?

*I’ll start with an example on Python 2.7. Python 3 coroutine is a logical continuation of older generator-like coroutines and based on the 2.7 implementation.* yield keyword stops the execution of current computation and suspends internal coroutine’s state. Call of send coroutine’s method awakes it, and the next iteration performed then again the state is suspended on yield keyword. While the coroutine is suspended control passed to other (co)routine, without losing its state. Also, send method could receive a single argument which will be available in a variable assigned to (yield) statement. Technically that is all you need to know to use Python 2 coroutines.

def coro(arg):

internal\_state = arg \*\* 2

while True:

awaited = (yield) # <<< Suspended here

internal\_state += awaited

print internal\_state

suspendable = coro(2) # Initialize coroutine

suspendable.send() # Start coroutine

suspendable.send(2) # Awake coroutine and pass argument to continue computation

Python 2.x coroutine basic usage

What is suspendability?

***Call stack.***

A stack is a **LIFO** abstract data type with two operations: push and pull. The last pushed item will be pulled out first. The call**stack**is a stack that contains call frames. And **call frame** is a data transmission unit which in its turn includes: *arguments*, *locals* and *return address* of an actual routine call.

*A caller pushes a return address into a call stack, and a called subroutine. When an operation is finished it pulls the return address off the call stack and transfers control to that address.*

If a call frame pulled out then all its locals and return address arguments went away with the frame. All the context is gone. What does routine need to switch control to another routine and then, possibly after several switches of control, continue computation? To be COroutine, it needs context! It needs to have precisely the same context of computation where control is passed last time.

Here we go to***the first-class continuation***.

*First-class continuation is ability to completely control the execution order of instructions. They can be used to jump to a function that produced the call to the current function, or to a function that has previously exited.****One can think of a first-class continuation as saving the*execution*state of the program.****It is important to note that true first-class continuations do not save program data it saves only the execution context.*

Python is a stack machine, and it implements its call stack on a much higher level of abstraction than a machine call stack. Python call stack is a part of Python virtual machine. It implements modern programming language features like closures, generators, and coroutines, and so on. In this article I’m focused on generators and coroutines but keep in mind — closures are pretty exciting and related to call stack topic too.

In Python C implementation source introduced a type PyFrameObject just like it said as “**frame object.”** Python uses it quite everywhere; also, **it is the first argument of** gen\_send\_ex **function which implements a coroutine** send method. When it goes to generators/coroutines PyFrameObject is managed in a slightly different manner — it keeps the state of computation (using value stack, last instruction e.t.c.) to suspend and resume coroutine/generator execution. So it *could be considered* as a first-class continuation object when it comes to coroutines. PyFrameObject implementation from a 3.6 branch of CPython listed below. It contains brilliant comments. Most of the comments describe how stack manipulated when it handled accordingly to generator purposes.

PyFrameObject implementation

typedef struct \_frame {

PyObject\_VAR\_HEAD

struct \_frame \*f\_back; /\* previous frame, or NULL \*/

PyCodeObject \*f\_code; /\* code segment \*/

PyObject \*f\_builtins; /\* builtin symbol table (PyDictObject) \*/

PyObject \*f\_globals; /\* global symbol table (PyDictObject) \*/

PyObject \*f\_locals; /\* local symbol table (any mapping) \*/

PyObject \*\*f\_valuestack; /\* points after the last local \*/

/\* Next free slot in f\_valuestack. Frame creation sets to f\_valuestack.

Frame evaluation usually NULLs it, but a frame that yields sets it

to the current stack top. \*/

PyObject \*\*f\_stacktop;

PyObject \*f\_trace; /\* Trace function \*/

/\* In a generator, we need to be able to swap between the exception

state inside the generator and the exception state of the calling

frame (which shouldn't be impacted when the generator "yields"

from an except handler).

These three fields exist exactly for that, and are unused for

non-generator frames. See the save\_exc\_state and swap\_exc\_state

functions in ceval.c for details of their use. \*/

PyObject \*f\_exc\_type, \*f\_exc\_value, \*f\_exc\_traceback;

/\* Borrowed reference to a generator, or NULL \*/

PyObject \*f\_gen;

int f\_lasti; /\* Last instruction if called \*/

/\* Call PyFrame\_GetLineNumber() instead of reading this field

directly. As of 2.3 f\_lineno is only valid when tracing is

active (i.e. when f\_trace is set). At other times we use

PyCode\_Addr2Line to calculate the line from the current

bytecode index. \*/

int f\_lineno; /\* Current line number \*/

int f\_iblock; /\* index in f\_blockstack \*/

char f\_executing; /\* whether the frame is still executing \*/

PyTryBlock f\_blockstack[CO\_MAXBLOCKS]; /\* for try and loop blocks \*/

PyObject \*f\_localsplus[1]; /\* locals+stack, dynamically sized \*/

} PyFrameObject;

Python source code fragment

How about Python ≥ 3.5?

Python 3.5 introduced native coroutines’ async await syntax which is further improvement of yield from asyncio.coroutinesyntaxintroduced in Python 3.4.In essence, coroutine kept pretty much the same. We can find an interesting comment in gen\_send\_ex the function of C implementation:

/\* `gen` is either:  
 \* a generator with CO\_FUTURE\_GENERATOR\_STOP flag; \* a coroutine;  
 \* a generator with CO\_ITERABLE\_COROUTINE flag (decorated with types.coroutine decorator);  
 \* an async generator. \*/

Below we will take new syntax apart and pile up an understanding of Python native coroutines.

async def coro():

return 2

async def coro2():

b = await coro()

return b \*\* 2

async def coro3():

c = await coro2()

return c \*\* 2

if \_\_name\_\_ == "\_\_main\_\_":

c = coro3()

c.send(None)

print(c.send(None))

Python 3.x native coroutine basic usage

async makes function a native coroutine, e.g., awaitable object, that is clear. Let me be opinionated d\*ck for a couple of seconds: I don’t like it! Because it is not asynchronous, it is suspendable, awaitable whatever but not async. Asynchrony is an interaction between the co-op multitasking scheduler and the computation process(es). Yes, it is a natural solution to use coroutines to implement a scheduler. But coroutine and asynchrony are different notions from different levels of abstraction. Typically I don’t care, but 8 of 10 people even involved in programming think that coroutines are asynchronous even without that word in front of coro declaration.

await

From a first glance pretty much the same as but it is not. And again I’m not a massive fan of this syntax. await keyword is incomplete. await could be used only before the awaitable object. Thus each coroutine needs another coroutine to await. Where is the end?

Native coroutine has two options dealing with await: first — await from the awaitable object, second — don’t have it at all. But the last one returns immediately. It is weird to define it as a coroutine if it can’t be like that to break the paradox.

The second thing you should note — you can return final result of computation from a coroutine. It feels like corruption of coroutines strict data consumption conception.

*• Keeping it Straight 33*

*• Generators produce data for iteration*

*• Coroutines are consumers of data*

*• To keep your brain from exploding, you don’t mix the two concepts together*

***David Beazley***

It makes things tangled, but in the end, it is comfortable. This return effective in send function, i.e. when object awaited or send called directly.

If you take a look at the clue difference between yield and await the existence of return becomes obvious. await is reversed yield. yield waits when send be called from the outer caller but await waits when awaited (callee) object return. Also send could be called directly (and that is already tangled).

Get a cup of tea…

**Because** send **still** **takes an argument!** I passed it, and where is it? In an example listed above, it is nowhere. Let’s build another one.

class Coro1(object):

def \_\_await\_\_(self):

a = yield # Obtain sent value

return a \*\* 2 # Return computed vaue back

async def coro2():

b = await Coro1() # Get return value from Coro1 instance

return b \*\* 2 # Return computed value further upward to pipeline

async def coro3():

c = await coro2() # Get return value from coro2

return c \*\* 2 # Return final value from pipeline

if \_\_name\_\_ == "\_\_main\_\_":

c = coro3()

c.send(None) # Start pipeline

print(c.send(2)) # Get final computation value

Python 3.x native coroutine argument usage

As it was mentioned, the control flow of native coroutines pipelines is reversed compared to Py2 coroutines. await suspends execution until awaitable object return. When you call send directly and pass an argument in it each await moves your argument to the next coroutine downward by pipeline until the last awaitable object in the pipeline is reached. The values returned from each awaitable will be passed upward to the pipeline. This arrangement should be taken into account when you build a pipeline with coroutines.

Note that we use class satisfying interface described in PEP 492: implements \_\_await\_\_ method that returns an iterator. And BOOM! Here we go with yield in \_\_await\_\_ method to obtain value sent from the pipeline initialization by send method.

Conclusion

A coroutine is not asynchronous but suspendable (awaitable) and has the right to exist outside of asyncio.

Python 3 extended syntax introduced in Python 2 and provides a higher level of abstraction for coroutines.

Python 3 coroutines implement different control flow than Python 2.

**PS**

I deliberately avoided topics of asynchrony and concurrency to focus on merely basic principles. Often the topics remain unobvious and create a mystical haze around coroutines.