Meet Generators Counterpart: Coroutines

Axel Sirota
MACHINE LEARNING ENGINEER
@AxelSirota

Generators Counterpart: Coroutines

A Generator Example

```
In [1]: def my_coroutine(a):
   ...:     print(f'--> Started with {a}')
   ...:     b = yield
   ...:     print(f'But continues with {b}')
   ...:
```

◀ **# Yield on the RIGHT SIDE**

Yield expression will yield every value sent to the coroutine and assign that value to the variable on the left

A Pipeline Example

```
In [2]: mycoro = my_coroutine(2)

In [3]: next(mycoro)
--> Started with 2

In [4]: mycoro.send(5)
But continues with 5
---------------------------------------------------------------------
StopIteration                             Traceback
(most recent call last)
<ipython-input-5-7eb8742afb11> in
<module>
----> 1 mycoro.send(5)

StopIteration:
```

- **# We instantiate the generator object**
- **# We need to advance to the first yield**
- **# Check that it assigned b to be 5!**
- **# Ends with Stop Iteration**

Send Default

```
In [2]: mycoro = my_coroutine(2)

In [3]: next(mycoro)
--> Started with 2

In [4]: next(mycoro)
But continues with None
---------------------------------------------------------------------
StopIteration                             Traceback
(most recent call last)
<ipython-input-5-7eb8742afb11> in
<module>
----> 1 mycoro.send(5)

StopIteration:
```

- **# We just iterate**
- **# It assigned None, so the default of send is __next__**

Checking the State

```
In [16]: mycoro = my_coroutine(2)

In [17]: inspect.getgeneratorstate(mycoro)
Out[17]: 'GEN_CREATED'

In [18]: next(mycoro)
--> Started with 2

In [19]: inspect.getgeneratorstate(mycoro)
Out[19]: 'GEN_SUSPENDED'

In [20]: mycoro.send(5)
But continues with 5
---------------------------------------------------------------------
------
StopIteration                             Traceback (most recent
call last)
<ipython-input-20-7eb8742afb11> in <module>
----> 1 mycoro.send(5)

StopIteration:

In [21]: inspect.getgeneratorstate(mycoro)
Out[21]: 'GEN_CLOSED'
```

◀ **# When created is GEN_CREATED**

◀ **# In the middle is GEN_SUSPENDED**

◀ **# When finished is GEN_CLOSED**

Closing as we wish

```
In [22]: mycoro = my_coroutine(2)

In [23]: inspect.getgeneratorstate(mycoro)
Out[23]: 'GEN_CREATED'

In [24]: mycoro.close()

In [25]: inspect.getgeneratorstate(mycoro)
Out[25]: 'GEN_CLOSED'

In [26]: next(mycoro)
---------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-26-567d0ccdf463> in <module>
----> 1 next(mycoro)

StopIteration:
```

◀ **# When created is GEN_CREATED**
◀ **# We can close whenever**

◀ **# When finished is GEN_CLOSED**

◀ **# If we try to iterate on that we get an Exception**

Expanding on Coroutines: Priming and Yielding

What happens if we send values to a CREATED coroutine?

Disaster!

```
In [27]: mycoro = my_coroutine(2)

In [28]: mycoro.send(5)
---------------------------------------------------------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-28-7eb8742afb11> in
<module>
----> 1 mycoro.send(5)
```

TypeError: can't send non-None value to a just-started generator

◀ **# We create a coroutine**

◀ **# Try to send values**

◀ **# Disaster!**

Priming Decorator

```
def coroutine(func):
    def start(*args, **kwargs):
        cr = func(*args, **kwargs)
        next(cr)
        return cr

    return start
```

◀ **# We create the coroutine**
◀ **# We call next on it automagically**
◀ **# return the primed coroutine**

```
@coroutine
def my_coroutine(a):
    print(f'--> Started with {a}')
    b = yield
    print(f'But continues with {b}')
```

◀ **#Super easy to use**

Priming Decorator

```
In [34]: mycoro = my_coroutine(2)
--> Started with 2

In [35]: mycoro.send(5)
But continues with 5
---------------------------------------------------------------------
StopIteration                             Traceback
(most recent call last)
<ipython-input-35-7eb8742afb11> in
<module>
----> 1 mycoro.send(5)

StopIteration:
```

- **# Check that at instantiation we already executed the print statement!**
- **# Try to send values and success**

Yielding Coroutines

```
def coroutine_yield(a):
    print(f'--> Started with {a}')
    b = yield a
    print(f'However I got sent a {b} and
yielded back {a}')
    c = a + b
    yield c
    print(f'I yielded back a {c} and exited')
```

◀ **# Not only we assign the sent value to b, but yield a? How?**

◀ **# Let's try to debug this with prints!**

Yielding Coroutines

```
In [8]: mycoro = coroutine_yield(2)

In [9]: gotten_first = next(mycoro)
--> Started with 2

In [10]: gotten_first
Out[10]: 2

In [11]: gotten_second = mycoro.send(5)
However I got sent a 5 and yielded back 2

In [12]: gotten_second
Out[12]: 7

In [13]: next(mycoro)
I yielded back a 7 and exited
```

◀ **# On the priming, we execute code up to the yield.**

◀ **# Check that we yielded 2! So, we executed up to. The right side!**

◀ **# As we can check in the print**

◀ **# We yielded 7 = 2 + 5**

◀ **# And exited!**

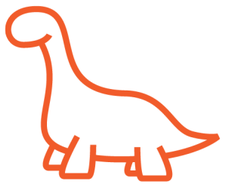
Yielding Coroutines

```
def coroutine_yield(a):
    print(f'--> Started with {a}')
    b = yield a
    print(f'However I got sent a {b} and yielded back {a}')
    c = a + b
    yield c
    print(f'I yielded back a {c} and exited')
```

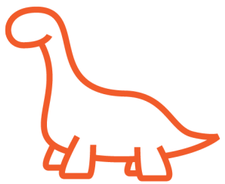
coro.send(b) → b = yield a → a = next(coro)

yield c → c = coro.send(b)

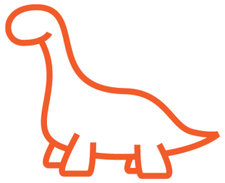
Citing The Master



Generators and coroutines are basically two different concepts



Generators produce data for iteration



Coroutines are consumers of data

Coroutines are just another concurrency model

Forever Coroutines

```
#  Example from Dave Beazley
@coroutine
def pattern(language):
    print(f'I only react to {language} messages')
    while True:
        line = yield
        try:
            if language in line:
                print(line)
        except TypeError:
            print('Please send me a string message')
```

◀ **# We wait for a line forever**

◀ **# If the pattern exists, it prints**

◀ **# We catch possible exceptions, we run forever**

Forever Coroutines

```
In [2]: checking = pattern('Python')
I only react to Python messages

In [3]: checking.send(None)
Please send me a string message

In [4]: checking.send('Okey')

In [5]: checking.send('Pluralsight is great!')

In [6]: checking.send('But Python rocks!')
But Python rocks!

In [7]: checking.close()
```

◀ **# We prime the coroutine**

◀ **# Doesn’t match the pattern**

◀ **# Same**

◀ **# Here it does!**

◀ **# We need to close it since it runs forever**

Closing a Running Coroutine

```
@coroutine
def my_slow_coroutine():
    print(f'I am so slow')
    yield
    sleep(10)
    print(f'But I am done!')


def run_coroutine(coro):
    try:
        coro.send(None)
    except StopIteration:
        print('Coroutine Done!')


mycoro = my_slow_coroutine()
thread = Thread(target=run_coroutine,
args=(mycoro, ))
thread.start()
mycoro.close()
thread.join()
print('Thread done!')
```

◀ **# We sleep to simulate a large calculation**

◀ **# Just triggers the coroutine**

◀ **# We try to close it**

Closing a Running Coroutine

Output:

```
I am so slow
Traceback (most recent call last):
  File "module4/
concurrency_coroutines.py", line 25, in
<module>
    mycoro.close()
ValueError: generator already executing
But I am done!
Coroutine Done!
```

◀ **# It started**

◀ **# We cannot close it!**

Cooperative vs Preemptive Multitasking

COOPERATIVE

Coroutines

PREEMPTIVE

Threads

Getting the Final State out of Generators: Returning Values

Coroutine Returns

```
@coroutine
def averager_with_result():
    Result = namedtuple('Result', ['Count',
'Average'])
    total = 0
    sum = 0
    average = None
    while True:
        value = yield average
        if value is None:
            break
        total += 1
        sum += value
        average = sum / total
    return Result(total, average)
```

◀ **# Each send we yield the average up to that point**

◀ **# We return the final value**

How does it work???

Coroutine Returns

```
In [2]: averager = module3.averager_with_result()

In [3]: averager.send(2)
Out[3]: 2.0

In [4]: averager.send(4)
Out[4]: 3.0

In [5]: averager.send(1)
Out[5]: 2.3333333333333335

In [6]: averager.send(5)
Out[6]: 3.0

In [7]: averager.send(None)
---------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-7-19d3349d9826> in <module>
----> 1 averager.send(None)

StopIteration: Result(Count=4, Average=3.0)
```

◀ **# We use None as sentinel value**

◀ **# The Result was returned as value of the StopIteration**

Generator Returns

```
def generator_with_return(size):
    magic_values =
random.random_integers(0, 10, size=size)
    for value in magic_values:
        yield value
    return magic_values
```

```
In [5]: for i in module3.generator_with_return(10):
   ...:     print(f'-->{i}')
   ...:
-->10
-->6
-->1
-->8
-->8
-->2
-->2
-->1
-->2
-->4
```

◀ **# We return the list**

◀ **# But we did not get it because of the for loop!**

Generator Returns

```
In [6]: def
generate_with_negated_and_data(negated,
data):
   ...:     result = namedtuple('Result', ['Data',
'Negated'])
   ...:     for d in data:
   ...:         for n in negated:
   ...:             yield n + d + 1
   ...:     return result(data, negated)
   ...:
   ...:
   ...: def pipeline(number):
   ...:     data = (i for i in range(number))
   ...:     squared = (i**2 for i in data)
   ...:     negated = (-i for i in squared)
   ...:     return
generate_with_negated_and_data(negated=negated, data=data)

In [11]: sum(pipeline(9))
Out[11]: -196
```

◀ **# We return the result**

◀ **# But again, for generators we are doomed because of the for loop magic**

Retrieving the Returned Value

```
In [20]: averager =
module4.averager_with_result()

In [21]: averager.send(3)
Out[21]: 3.0

In [22]: averager.send(5)
Out[22]: 4.0

In [23]: try:
    ...:     averager.send(None)
    ...: except StopIteration as exc:
    ...:     returned_value = exc.value
    ...:

In [24]: returned_value
Out[24]: Result(Count=2, Average=4.0)
```

◀ **# No need to prime**

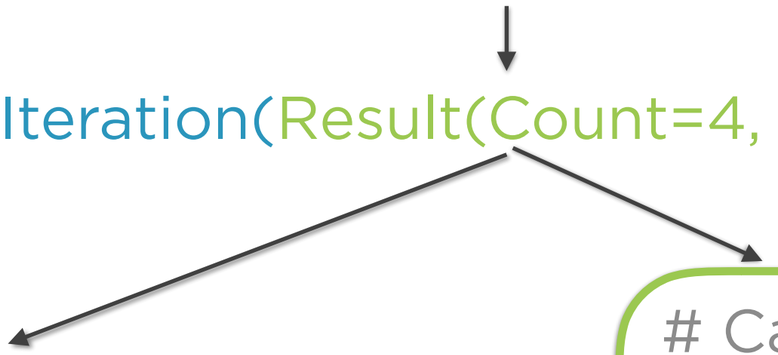
◀ **# Manually we force the return and retrieve the value of the exception**

◀ **# Therefore getting Result back**

Retrieving the Returned Value

```
def coroutine_with_return(a):
    # .. stuff..

    return Result(Count=4, Average=3.5)
```



```
StopIteration(Result(Count=4, Average=3.5))
```

```
# Caller
for i in
coroutine_with_return(VALUE):
    # stuff...
```

```
# Caller
my_coro =
coroutine_with_return(VALUE)
try:
    mycoro.send(None)
except StopIteration as exc:
    returned_value = exc.value
```

Yield Data Model:
Iter, Next, Send, Close, Throw and Return

Special Methods

iter

next

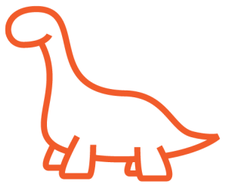
send

close

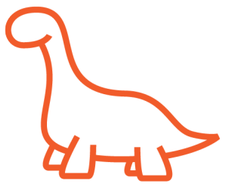
throw

return

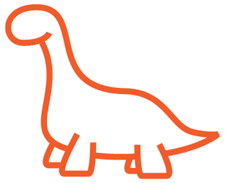
Ways of Coroutine Ending



Raise an internal exception



Closed from the caller



Thrown exception from the caller

Exception Handling

```
@coroutine
def coroutine_exception(number):
    print('-> coroutine started')
    while True:
        try:
            x = yield
        except ValueError:
            print('*** ValueError handled. Continuing...')
        except GeneratorExit:
            print('This is executed if I get closed, so I need to cleanup here and die gracefully')
            raise
        else:
            print('-> coroutine received: {!r}'.format(x))
            number + x
```

◀ **# Get a value sent**

◀ **# If we get thrown a ValueError we enter this block**

◀ **# If we get closed we enter this block. We must reraise the exception**

◀ **# After we got sent a value, print it**

◀ **# Force possible exceptions inside by assuming x allows __sum__**

Exception From Inside

```
In [3]: mycoro.send(2)
-> coroutine received: 2

In [4]: mycoro.send(5)
-> coroutine received: 5

In [5]: mycoro.send(None)
-> coroutine received: None
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent
call last)
<ipython-input-5-db1ac2c02de8> in <module>
----> 1 mycoro.send(None)

<ipython-input-1-2b8685773b52> in
coroutine_exception(number)
     14         else:
     15             print('-> coroutine received: {!r}'.format(x))
---> 16             number + x

TypeError: unsupported operand type(s) for +: 'int' and
'NoneType'
```

◀ **# We send None, that doesn't allow adds**

◀ **# Easy piece, we throw the exception to the caller.**

Exception From Inside

```
In [14]: mycoro = coroutine_exception(15)
-> coroutine started

In [15]: try:
    ...:     mycoro.send(None)
    ...: except TypeError:
    ...:     print('I caught you man!')
    ...:     print(
             inspect.getgeneratorstate(mycoro)
)
    ...:
-> coroutine received: None
I caught you man!
GEN_CLOSED
```

◀ **# We catch the exception!**

◀ **# We check the state of the coroutine after catching the exception**

◀ **# But it closed itself!**

Closing the Coroutine

```
In [29]: mycoro = coroutine_exception(2)
-> coroutine started

In [30]: mycoro.close()
This is executed if I get closed, so I need
to cleanup here and die gracefully

In [31]: inspect.getgeneratorstate(mycoro)
Out[31]: 'GEN_CLOSED'
```

◀ **# We close the coroutine**
◀ **# Our code inside the block was executed**

◀ **# Now the coroutine is closed!**

Closing the Coroutine

```
@coroutine
def coroutine_without_reraise():
    while True:
        try:
            x = yield
        except GeneratorExit:
            print('I do nothing')
        else:
            print(f'Got value {x}')

In [39]: mycoro = coroutine_wihtout_reraise()

In [40]: mycoro.send(2)
Got value 2

In [41]: mycoro.close()
I do nothing
---------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-42-933fee6d5d2d> in <module>
----> 1 mycoro.close()

RuntimeError: generator ignored GeneratorExit
```

◀ **# If we don’t reraise the exception and end**

◀ **# The block gets executed**

◀ **# Followed by a RuntimeError that cannot be caught**

Exception Thrown To the Coroutine

```
In [44]: mycoro.throw(ValueError)
*** ValueError handled. Continuing...

In [45]: mycoro.send(2)
-> coroutine received: 2

In [46]: mycoro.throw(TypeError)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
call last)
<ipython-input-46-c60a475d7cf5> in <module>
----> 1 mycoro.throw(TypeError)

<ipython-input-28-c006264a7764> in
coroutine_exception(number)
      7     while True:
      8         try:
----> 9             x = yield
     10         except ValueError:
     11             print('*** ValueError handled. Continuing...')

TypeError:
```

◀ **# If we throw the exception that was handled**

◀ **# It executed the block and continued!**

◀ **# But if we throw another exception**

◀ **# Of course, it dies...**

A Caveat

```
In [47]: mycoro.throw(ValueError('Oh oh'))
---------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-d6e39fcf3eb4> in <module>
----> 1 mycoro.throw(ValueError('Oh oh'))

ValueError: Oh oh
```

◀ **# Always send the Exception class, NOT an instance of it**

◀ **# Those will NEVER get caught**

We have 6 methods in the yield data model

Close and throw finish the coroutines early

If we close, a GeneratorExit is raised. This cannot be ignored

If we throw, we can catch the Exception class and continue

Demo

Recreate our average solution with coroutines

Use priming and yielding to get intermediate averages

Use returns to fetch end state

Manage exceptions and closing accordingly

Summary

Coroutines get sent values

We need to prime coroutines until the first yield

Coroutines can also yield intermediate state apart from be sent values

Coroutines can return values encapsulated as the value of the StopIteration

Also we can close or throw exceptions to coroutines and they can handle them