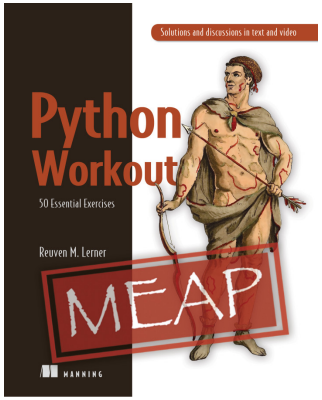
Generators, coroutines, and nanoservices

Reuven M. Lerner • PiterPy 2020
reuven@lerner.co.il • @reuvenmlerner

I teach Python!

- Corporate training
- Video courses about Python + Git
- Weekly Python Exercise
  - More info at https://lerner.co.il/

- "Python Workout" — published by Manning
- https://PythonWorkout.com



- "Better developers" — free, weekly newsletter about Python
  - https://BetterDevelopersWeekly.com/

This is not an “asyncio” talk

The dumbest function in the world

```
def myfunc():
    return 1
    return 2
    return 3


print(myfunc())
```

- What happens when we run this code?

```
1
```

Not a surprise!

```
def myfunc():
    return 1
    return 2
    return 3
```

- This makes sense, after all:
  - "return" means: stop the function, and return a value
  - "pylint" warns us that the final lines are "unreachable"
  - Even Python's byte compiler ignores the final two lines!

Bytecode from our function

```
>>> import dis
>>> dis.dis(myfunc)

  2           0 LOAD_CONST               1 (1)
              2 RETURN_VALUE
```

What about this function?

```
def mygen():
    yield 1
    yield 2
    yield 3


print(mygen())
```

- What happens when we run this code?

```
<generator object mygen at 0x10bdc7660>
```

What's the difference?

- In a nutshell: "yield" makes the difference!
- The use of "yield" turns a regular function into a *generator function*.
- When Python compiles your function, it notices the "yield" and tags the function as a generator function.
- Running a "generator function" returns a "generator."

Seeing this with dis.show_code

```
>>> dis.show_code(mygen)

Name:                mygen
Filename:            ./gen3.py
Argument count:      0
Positional-only arguments: 0
Kw-only arguments: 0
Number of locals:  0
Stack size:          1
Flags:               OPTIMIZED, NEWLOCALS, GENERATOR, NOFREE
Constants:
   0: None
   1: 1
   2: 2
   3: 3
```

Bytecodes

```
>>> dis.dis(mygen)

  2           0 LOAD_CONST               1 (1)
              2 YIELD_VALUE
              4 POP_TOP

  3           6 LOAD_CONST               2 (2)
              8 YIELD_VALUE
             10 POP_TOP

  4          12 LOAD_CONST               3 (3)
             14 YIELD_VALUE
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

What's a generator?

- It implements Python's "iterator" protocol:
  - You get its iterator via "iter" (which is itself)
  - You get each succeeding value with "next"
  - When it gets to the end, it raises "StopIteration"

How does a "for" loop work?

```
for one_item in 'abcd':
    print(one_item)
```

(1) Is it iterable?

(2) If so, then repeatedly ask for the next thing, assign to one_item, and run the loop body

(3) When the object raises StopIteration, exit the loop

```
a
b
c
d
```

What about with our generator?

```
for one_item in mygen():
    print(one_item)
```

(1) Is it iterable?

(2) If so, then repeatedly ask for the next thing, assign to one_item, and run the loop body

(3) When the object raises StopIteration, exit the loop

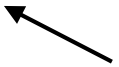
```
1
2
3
```

Doing it manually

```
>>> g = mygen()


>>> g
<generator object mygen at 0x111083120>

>>> iter(g)
<generator object mygen at 0x111083120>
```



Same address,

Same ID

What about next?

```
>>> next(g)
1


>>> next(g)
2


>>> next(g)
3


>>> next(g)
StopIteration:
```

```
def mygen():
    yield 1
    yield 2
    yield 3

g = mygen()
```

“next” and generators

- “next” tells a generator to run through the next “yield” statement
- This can be a lot of code, or a tiny bit of code
- Upon hitting “yield”, the generator returns the value and goes to sleep
- The function’s state remains across those calls

Generator example 1: fib

```
def fib():
    first = 0
    second = 1

    while True:
        yield first
        first, second = second, first+second

>>> g = fib()
>>> next(g)   # 0
>>> next(g)   # 1
>>> next(g)   # 1
>>> next(g)   # 2
```

Don't do this!

```
list(fib())
```

- The call to "list" will wait to get StopIteration...
- ... which won't ever happen, of course

Generator example 2: read_n

```
def read_n(filename, n):
    f = open(filename)

    while True:
        output = ''.join(f.readline()
                         for i in range(n))
        if output:
            yield output
        else:
            break
```

Generator example 3: next_vowel

```
def get_vowels(filename):
    for one_line in open(filename):
        for one_char in one_line:
            if one_char.lower() in 'aeiou':
                yield one_char
```

So, when do we use generators?

- We have a potentially large (or infinite) set of values to return
- It's easier to express the idea as a function
- We have to set things up (e.g., a network connection or file)
- We want to keep state across runs in local variables

What about this?

```
def mygen():
    x = None
    while True:
        x = yield x
        x *= 5
```

“yield” as an expression

- The rules for generators still apply
  - Each call to “next” runs the generator through the next “yield”
  - After “yield”, it goes to sleep
- But “yield” now allows for two-way communication!
- It can receive a message from the outside world
- The received value replaces “yield”, typically in assignment

The "send" method

- We can advance a generator with the "next" function:

```
next(g)
```

- We can *send a value* to a generator with the "send" method:

```
g.send(123)
```

- Note: A call to next(g) is the same as g.send(None)

Using our generator with "send"

```
def mygen():
    x = None
    while True:
        x = yield x
        x *= 5
```

```
g = mygen()
next(g)                 # prime it
g.send(10)              # 50
g.send(23)              # 115
g.send('abc')           # abcabcabcabcabc
g.send([1, 2, 3])       # [1,2,3,1,2,3,1,2,3 …]
```

This is a “coroutine”

- A coroutine is:
  - A generator
  - It waits to get input from elsewhere using “send”
  - The data is received with “yield” as an expression
  - (typically on the right side of an assignment)
  - Local state remains across calls

Walrus + yield

- Python 3.8 introduced the "assignment expression" operator
- :=
- Very controversial!
- And yet, in these circumstances, quite useful:

```
def mygen():
    x = None
    while x := (yield x):
        x *= 5
```

Exit the loop when we get None or any empty value

Put "yield x" in parens for it to work

Don't want to prime it?

- David Beazley, in a great 2009 talk, suggested using a decorator
- This decorator automatically runs "next", and then returns the already-primed coroutine
- Now you just need to use "send" on it
- The point of priming, of course, is to move things ahead such that you can send (giving "yield" a value)

So what?

How can I use coroutines?

- Does this seem like a solution looking for a problem?
  - If you think so, you're not alone.
- This talk, though is mean to give you some ideas for how we can use coroutines.

Coroutine as "nanoservices"

- Many applications use a "microservice" architecture
  - That is: You divide your app into different parts
  - Each part resides in a different server
  - You access each "microservice" via a distinct API
- I like to think of coroutines as "nanoservices"
  - Very small, in-memory services
  - No network, object, thread, or process overhead
  - Always running, keeping its state
- You'll need to create your own protocol for a coroutine
  - But you can use all of Python's data structures

Example: MD5

- MD5 is a popular hash function, available in "hashlib"
- It's a bit of a pain to use it in Python
- Let's create a coroutine that'll provide an MD5 service!

Coroutine example 1: MD5

```
import hashlib

def md5_gen():
    output = None

    while s := (yield output):
        m = hashlib.md5()
        m.update(s.encode())
        output = m.hexdigest()

>>> g = md5_gen()
>>> g.send(None)
>>> print(g.send('hello'))
>>> print(g.send('goodbye'))
>>> g.send(None)                    # stop the service
```

Coroutine example 2: Weather

```
import requests


def get_forecasts(city):
    weather = requests.get(
        f'https://worldweather.wmo.int/en/json/{city}_en.json').json()
    for one_forecast in weather['city']['forecast']['forecastDay']:
        yield one_forecast


>>> g = get_forecasts(44)

>>> print(next(g))
{'forecastDate': '2020-07-31', 'wxdesc': '', 'weather': 'Humid', 'minTemp': '26',
'maxTemp': '31', 'minTempF': '79', 'maxTempF': '88', 'weatherIcon': 2702}

>>> print(next(g))
{'forecastDate': '2020-08-01', 'wxdesc': '', 'weather': 'Humid', 'minTemp': '24',
'maxTemp': '32', 'minTempF': '75', 'maxTempF': '90', 'weatherIcon': 2702}

>>> print(next(g))
{'forecastDate': '2020-08-02', 'wxdesc': '', 'weather': 'Clear', 'minTemp': '25',
'maxTemp': '31', 'minTempF': '77', 'maxTempF': '88', 'weatherIcon': 2502}
```

Wait! That's not a coroutine!

Coroutine example 2.1: Weather

```
import requests

def get_forecasts():

    while city_id := (yield 'Send a city number or None'):

        weather = requests.get(
            f'https://worldweather.wmo.int/en/json/{city_id}_en.json').json()
        for one_forecast in weather['city']['forecast']['forecastDay']:
            yield one_forecast

>>> g = get_forecasts()

>>> next(g)
'Send a city number or None'

>>> print(g.send(44))
{'forecastDate': '2020-07-31', 'wxdesc': '', 'weather': 'Humid', 'minTemp': '26', 'maxTemp':
'31', 'minTempF': '79', 'maxTempF': '88', 'weatherIcon': 2702}

>>> print(g.send(44))
{'forecastDate': '2020-08-01', 'wxdesc': '', 'weather': 'Humid', 'minTemp': '24', 'maxTemp':
'32', 'minTempF': '75', 'maxTempF': '90', 'weatherIcon': 2702}

>>> print(g.send(44))
{'forecastDate': '2020-08-02', 'wxdesc': '', 'weather': 'Clear', 'minTemp': '25', 'maxTemp':
'31', 'minTempF': '77', 'maxTempF': '88', 'weatherIcon': 2502}
```

Continued...

```
>>> print(g.send(44))
{'forecastDate': '2020-08-03', 'wxdesc': '', 'weather': 'Partly
Cloudy', 'minTemp': '26', 'maxTemp': '31', 'minTempF': '79',
'maxTempF': '88', 'weatherIcon': 2202}


>>> print(g.send(44))
Send a city number or None


>>> print(g.send(45))
{'forecastDate': '2020-07-31', 'wxdesc': '', 'weather': 'Hot',
'minTemp': '26', 'maxTemp': '38', 'minTempF': '79', 'maxTempF': '100',
'weatherIcon': 3101}


>>> print(g.send(45))
{'forecastDate': '2020-08-01', 'wxdesc': '', 'weather': 'Hot',
'minTemp': '24', 'maxTemp': '39', 'minTempF': '75', 'maxTempF': '102',
'weatherIcon': 3101}
```

Coroutine example 3

```
import psycopg2


def people_api():

    conn = psycopg2.connect("dbname=people user=reuven")
    output = 'Send a query, or None to quit: '

    while d := (yield output):
        cur = conn.cursor()
        query = '''SELECT id, first_name, last_name, birthdate
        FROM People '''
        args = ()

        for field in ['first_name', 'last_name', 'birthdate']:
            if field in d:
                query += f' WHERE {field} = %s '
                args += (d[field],)

        print(query)

        cur.execute(query, args)
        for one_record in cur.fetchall():
            yield one_record
```

Now we can query our DB!

```
>>> g = people_api()
>>> next(g)
'Send a query, or None to quit: '

>>> g.send({'last_name':'Lerner'})
SELECT id, first_name, last_name, birthdate
        FROM People  WHERE last_name = %s
(1, 'Reuven', 'Lerner', datetime.datetime(1970, 7, 14, 0, 0))

>>> g.send('whatever')
'Send a query, or None to quit: '

>>> g.send({'last_name':'Lerner-Friedman'})
SELECT id, first_name, last_name, birthdate
        FROM People  WHERE last_name = %s
(2, 'Atara', 'Lerner-Friedman', datetime.datetime(2000, 12, 16, 0, 0))
>>> g.send('next')
(3, 'Shikma', 'Lerner-Friedman', datetime.datetime(2002, 12, 17, 0, 0))
>>> g.send('next')
(4, 'Amotz', 'Lerner-Friedman', datetime.datetime(2005, 10, 31, 0, 0))
>>> g.send('next')
'Send a query, or None to quit: '
```

Ending early

- How can I tell the generator that I'm completely done?
  - Use the "close" method
  - This exits from the generator without raising a StopIteration exception
  - If you try to use the generator after closing it, you'll get a StopIteration exception
- But what if I want to exit from the current query, keeping the generator around?
- It would be nice if we could send the generator a signal, telling it that we want to give it a new query

The “throw” method

- We can raise an exception in a generator using “throw”
- Remember that exceptions aren’t only for errors!
- You’ll almost certainly want to throw a custom exception

Revisiting the weather

```
import requests


class DifferentCityException(Exception):
    pass


def get_forecasts():

    while city_id := (yield 'Send a city number or None'):

        weather = requests.get(
            f'https://worldweather.wmo.int/en/json/{city_id}_en.json').json()

        try:
            for one_forecast in weather['city']['forecast']['forecastDay']:
                yield one_forecast
        except DifferentCityException:
            continue
```

Let's try it!

```
>>> g = get_forecasts()
>>> next(g)
'Send a city number or None'


>>> print(g.send(44))
{'forecastDate': '2020-08-01', 'wxdesc': '', 'weather': 'Humid', 'minTemp': '26',
'maxTemp': '32', 'minTempF': '79', 'maxTempF': '90', 'weatherIcon': 2702}


>>> print(g.send(44))
{'forecastDate': '2020-08-02', 'wxdesc': '', 'weather': 'Clear', 'minTemp': '24',
'maxTemp': '31', 'minTempF': '75', 'maxTempF': '88', 'weatherIcon': 2502}

>>> g.throw(DifferentCityException)
'Send a city number or None'

>>> print(g.send(49))
{'forecastDate': '2020-08-01', 'wxdesc': '', 'weather': 'Hot', 'minTemp': '25',
'maxTemp': '36', 'minTempF': '77', 'maxTempF': '97', 'weatherIcon': 3101}
```

Our nanoservice

- Available via a simple request-response interface
- Uses Python data structures for input and output
- Keeps state across invocations
  - Great for network and database connections
- Can cache data
  - Yields one element at a time
  - (Or your API can specify how many you want with each iteration)

Megaservices

- What if we want to have tons of functionality in our coroutine?
- For example, imagine if we want to have both MD5 and weather forecasting in the same coroutine.

- (Note: This is a bad idea, but stick with me!)

```
import requests
import hashlib


class DifferentCityException(Exception):
    pass


def combined_generator():
    while s := (yield 'Send 1 for weather, 2 for MD5, or None to exit'):
        if s == 1:
            while city_id := (yield 'Send a city number or None'):

                weather = requests.get(
                    f'https://worldweather.wmo.int/en/json/{city_id}_en.json').json()
                try:
                    for one_forecast in weather['city']['forecast']['forecastDay']:
                        yield one_forecast
                except DifferentCityException:
                    continue

        elif s == 2:
            output = 'Enter text to hash, or None'
            while s := (yield output):
                m = hashlib.md5()
                m.update(s.encode())
                output = m.hexdigest()

        else:
            output = 'Unknown choice; try again'
```

It works!

```
>>> g = combined_generator()

>>> next(g)

'Send 1 for weather, 2 for MD5, or None to exit'


>>> g.send(1)

'Send a city number or None'


>>> print(g.send(44))

{'forecastDate': '2020-08-02', 'wxdesc': '', 'weather': 'Clear', 'minTemp': '23',
'maxTemp': '31', 'minTempF': '73', 'maxTempF': '88', 'weatherIcon': 2502}


>>> g.throw(DifferentCityException)

'Send a city number or None'


>>> print(g.send(48))

{'forecastDate': '2020-08-03', 'wxdesc': '', 'weather': 'Sunny Periods', 'minTemp':
'17', 'maxTemp': '30', 'minTempF': '63', 'maxTempF': '86', 'weatherIcon': 2201}
```

Continued

```
>>> g.throw(DifferentCityException)
'Send a city number or None’

>>> print(g.send(None))
Send 1 for weather, 2 for MD5, or None to exit

>>> g.send(2)
'Enter text to hash, or None'

>>> g.send('hello')
'5d41402abc4b2a76b9719d911017c592'

>>> g.send('hello!')
'5a8dd3ad0756a93ded72b823b19dd877'

>>> g.send(None)
'Send 1 for weather, 2 for MD5, or None to exit'
```

Unfortunately, this works

- How can we improve this super-ugly code?
- If this were a function, we would break it down into smaller functions, and call those from the main function
- Why not try that with our generator?

Refactoring our generator

```
def combined_generator():
    while s := (yield 'Send 1 for weather, 2 for MD5, or None to exit'):

        if s == 1:
            city_generator()

        elif s == 2:
            md5_generator()

        else:
            output = 'Unknown choice; try again'


def city_generator():

    while city_id := (yield 'Send a city number or None'):

        weather = requests.get(
            f'https://worldweather.wmo.int/en/json/{city_id}_en.json').json()
        try:
            for one_forecast in weather['city']['forecast']['forecastDay']:
                yield one_forecast
        except DifferentCityException:
            continue


def md5_generator():
    output = 'Enter text to hash, or None'
    while s := (yield output):
        m = hashlib.md5()
        m.update(s.encode())
        output = m.hexdigest()
```

This doesn't work, though

```
>>> g = combined_generator()


>>> next(g)
'Send 1 for weather, 2 for MD5, or None to exit'


>>> g.send(1)
>>> 'Send 1 for weather, 2 for MD5, or None to
exit'


>>> g.send(1)
'Send 1 for weather, 2 for MD5, or None to exit'
```

What's wrong?

- Our "combined_generator" is just a regular function
- We need to yield something
- Maybe we can just iterate over the items that the generator returns?
- No. That won't be enough.
  - We are sending to the outside generator
  - It's the inside generator that needs to get the message
  - And what if we run g.throw? Or g.close?

"yield from" to the rescue!

```
def combined_generator():
    while s := (yield 'Send 1 for weather, 2 for MD5, or None to exit'):

        if s == 1:
            yield from city_generator()

        elif s == 2:
            yield from md5_generator()

        else:
            output = 'Unknown choice; try again'
```

What "yield from" does

- It isn't just running a "for" loop on the generator
  - We don't need a new keyword (as of 3.3) for that!
- Rather, "yield from" provides bidirectional communication with a coroutine
  - Sending to the outer is passed to the inner
  - Data yielded by the inner is passed to the outer
  - Exceptions raised via "throw" work
  - We can close the inner one via "close"
- In other words: "yield from" is made for sub-coroutines

```
>>> g = combined_generator()
>>> next(g)
'Send 1 for weather, 2 for MD5, or None to exit'

>>> g.send(2)
'Enter text to hash, or None'

>>> g.send('hello')
'5d41402abc4b2a76b9719d911017c592'

>>> g.send(None)
'Send 1 for weather, 2 for MD5, or None to exit'

>>> g.send(1)
'Send a city number or None'

>>> print(g.send(44))
{'forecastDate': '2020-08-02', 'wxdesc': '', 'weather': 'Clear', 'minTemp': '23', 'maxTemp':
'31', 'minTempF': '73', 'maxTempF': '88', 'weatherIcon': 2502}

>>> g.throw(DifferentCityException)
'Send a city number or None'
```

What about asyncio?

- Early versions of asyncio used generator functions
  - They can be stopped and started
  - Cooperative multitasking!
- Modern asyncio uses special keywords, such as “async def” and “await”
  - The ideas are similar, but implementations are different
  - Don’t be confused!

Should you use coroutines?

- They can be very useful!
  - Speedy, in-memory, arbitrary API
  - Provide us with large data, one chunk at a time
  - We can divide them into smaller pieces and use “yield from”
- But:
  - They’re not so well understood
  - It might seem weird to be using “send” in this way
  - (You might want to wrap it in a clearer API)
  - When things go wrong, it can be hard to debug

Questions or comments?

- Ask me here!

- Follow me on Twitter, @reuvenmlerner
- Follow me on YouTube
- Get free, weekly Python tips
  - https://BetterDevelopersWeekly.com/