Demystifying Coroutines and Asynchronous Programming in Python

Mariano Anaya
@rmarianoa

FOSDEM 2019 - Feb 03

History

- PEP-255: Simple generators
- PEP-342: Coroutines via enhanced generators
- PEP-380: Syntax for delegating to a sub-generator
- PEP-492: Coroutines with async and await syntax

Generators

PEP 255 -- Simple Generators

PEP:	255
Title:	Simple Generators
Author:	nas at arctrix.com (Neil Schemenauer), tim.peters at gmail.com (Tim Peters), magnus at hetland.org (Magnus Lie Hetland)
Discussions-To:	python-iterators at lists.sourceforge.net
Status:	Final
Type:	Standards Track
Requires:	234
Created:	18-May-2001
Python-Version:	2.2
Post-History:	14-Jun-2001, 23-Jun-2001

Generate elements, one at the time, and *suspend...*

- Save memory
- Support iteration pattern, infinite sequences, etc.

Simple Generators

- `next()` will advance until the next **yield**
    - Produce a value, & *suspend*
    - End? ⟶ `StopIteration`

Coroutines

PEP 342 -- Coroutines via Enhanced Generators

PEP:	342
Title:	Coroutines via Enhanced Generators
Author:	Guido van Rossum, Phillip J. Eby
Status:	Final
Type:	Standards Track
Created:	10-May-2005
Python-Version:	2.5
Post-History:	

Can simple generators…

- … suspend? ✔
- … send/receive data from the context? ✘
- … handle exceptions from the caller’s context? ✘

Generators as coroutines

New methods!

```
<g>.send(<value>)
<g>.throw(<exception>)
<g>.close()
```

Coroutines via Enhanced Generators

Coroutines are *syntactically* like generators.

Syntactically equivalent, semantically different.

Coroutines via Enhanced Generators

With `.send()`, the caller sends (receives) data to (from) the coroutine.

```
value = yield result
```

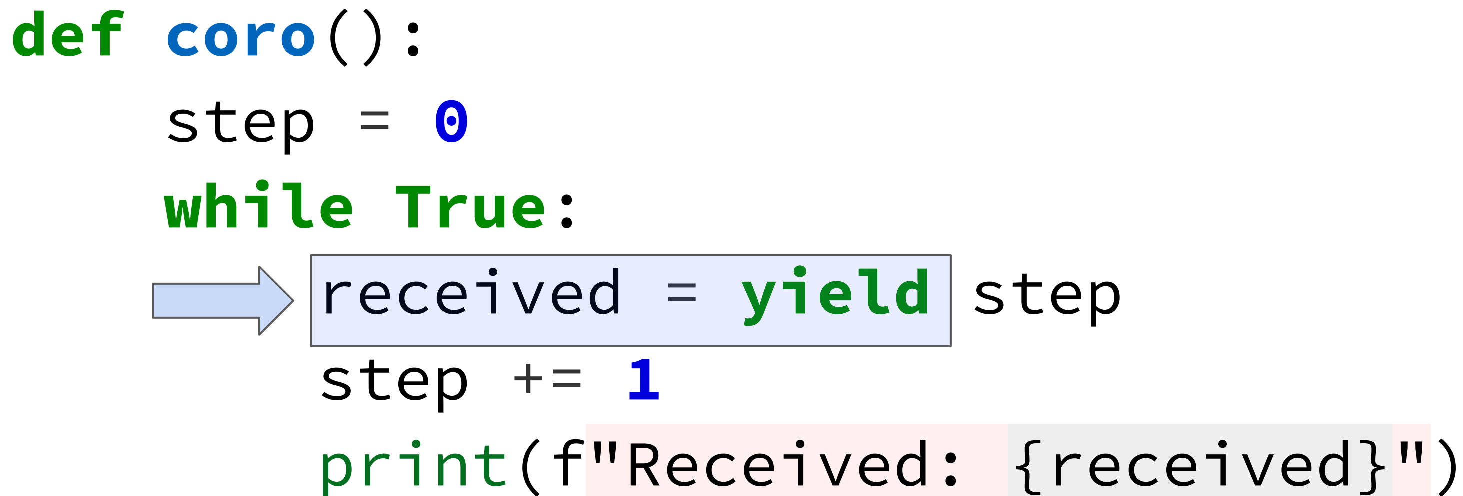
```python
def coro():
    step = 0
    while True:
        received = yield step
        step += 1
        print(f"Received: {received}")
```

```
>>> c = coro()
>>> next(c)
>>> step = c.send(received)
```

```
>>> c = coro()
>>> next(c)  # important!
0
```

```
def coro():
    step = 0
    while True:
        received = yield step
        step += 1
        print(f"Received: {received}")
```

```
>>> step = c.send(100)
```



```
>>> step = c.send(100)
Received: 100
```

```python
def coro():
    step = 0
    while True:
        received = yield step
        step += 1
        print(f"Received: {received}")
```

```
>>> step = c.send(100)
Received: 100
>>> step
1
```

```
def coro():
    step = 0
    while True:
        received = yield step
        step += 1
        print(f"Received: {received}")
```

```
>>> c.throw(ValueError)
---------------------
ValueError
Traceback (most recent call last)
----> 1 step = c.throw(ValueError)
      5     step = 0
      6     while True:
----> 7         received = yield step
      8         step += 1
      9         print(f"Received: {received}")
```

Can we do better?

Better Coroutines

PEP 380 -- Syntax for Delegating to a Subgenerator

PEP:	380
Title:	Syntax for Delegating to a Subgenerator
Author:	Gregory Ewing <greg.ewing at canterbury.ac.nz>
Status:	Final
Type:	Standards Track
Created:	13-Feb-2009
Python-Version:	3.3
Post-History:	
Resolution:	https://mail.python.org/pipermail/python-dev/2011-June/112010.html

Delegating to a Sub-Generator

- Enhancements
  - Generators can now `return` values!
  - `yield from`

Generators - Return values

→ `StopIteration.value`

```
>>> def gen():
...:     yield 1
...:     yield 2
...:     return 42
```

```
>>> g = gen()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
---------------------------
StopIteration
Traceback (most recent call last)
StopIteration: 42
```

`yield from` - Basic

Something in the form

```
yield from <iterable>
```

Can be thought of as

```
for e in <iterable>:
    yield e
```

`yield from` - More

- Nested coroutines: `.send()`, and `.throw()` are passed along.
- Capture return values

```
value = yield from coroutine(...)
```

`yield from` Example

```python
def internal(name, start, end):
    for i in range(start, end):
        value = yield i
        print(f"{name} got: {value}")
    print(f"{name} finished at {i}")
    return end

def general():
    start = yield from internal("first", 1, 5)
    end = yield from internal("second", start, 10)
    return end
```

```
>>> g = general()
>>> next(g)
```

```
>>> g = general()
>>> next(g)
1
```

```
>>> g = general()
>>> next(g)
1

>>> g.send("1st value sent to main coroutine")
```

```
>>> g = general()
>>> next(g)
1

>>> g.send("1st value sent to main
coroutine")
first got: 1st value sent to main
coroutine
2
```

```
...

>>> next(g)
first got: None
first finished at 4
5
```

```
...

>>> g.send("value sent to main coroutine")
second got: value sent to main coroutine
6
```

`yield from` - Recap

- Better way of combining generators/coroutines.
- Enables chaining generators and many iterables together.

Issues & limitations

async def / await

PEP 492 -- Coroutines with async and await syntax

PEP:	492
Title:	Coroutines with async and await syntax
Author:	Yury Selivanov <yury at magic.io>
Discussions-To:	<python-dev at python.org>
Status:	Final
Type:	Standards Track
Created:	09-Apr-2015
Python-Version:	3.5
Post-History:	17-Apr-2015, 21-Apr-2015, 27-Apr-2015, 29-Apr-2015, 05-May-2015

yield from → await

```python
# py 3.4
@asyncio.coroutine
def coroutine():
    yield from asyncio.sleep(1)

# py 3.5+
async def coroutine():
    await asyncio.sleep(1)
```

`await`

~ `yield from`, except that:

- Doesn't accept generators that aren't coroutines.
- Accepts *awaitable* objects
  - `__await__()`

asyncio

- Event loop → scheduled & run coroutines
  - Update them with `send()`/`throw()`.
- The coroutine we write, delegates with **`await`**, to some other 3rd party generator, that will do the actual I/O.
- Calling **`await`** gives the control back to the scheduler.

Summary

- Coroutines evolved from generators, but they're conceptually different ideas.
- `yield from` → `await`: more powerful coroutines (&types).
- A chain of `await` calls ends with a `yield`.

Thank You!

Mariano Anaya

@rmarianoa