Pushing Data through Pipelines with Coroutines

Axel Sirota
MACHINE LEARNING ENGINEER
@AxelSirota

Much More Than Just Iteration: Generator Based Data Pipeline

A Pipeline Example

```
In [9]: def pipeline(number):
   ...:     data = (i for i in range(number))
   ...:     squared = (i**2 for i in data)
   ...:     negated = (-i for i in squared)
   ...:     return (n + 1 for n in negated)
   ...:

In [10]: list(pipeline(10))
Out[10]: [1, 0, -3, -8, -15, -24, -35, -48, -63,
-80]
```

- # This is **NOT** a generator, but a **generator factory**
- # Get a generator that yields ints; this is range
- # Square each element of data *when needed*
- # Negate each element in squared.
- # Add 1 to each element. This is -X^2 + 1

 # Note that until we iterate over this generator, we don't execute!

A Pipeline Example

```
In [15]: def squared(iterable):
    ...:     return (i**2 for i in iterable)
    ...:

In [16]: def negated(iterable):
    ...:     return (-i for i in iterable)
    ...:

In [17]: def add_one(iterable):
    ...:     return (i + 1 for i in iterable)
    ...:

In [18]: def pipeline(number):
    ...:     return
add_one(negated(squared(range(number))))
    ...:

In [19]: list(pipeline(10))
Out[19]: [1, 0, -3, -8, -15, -24, -35, -48, -63, -80]
```

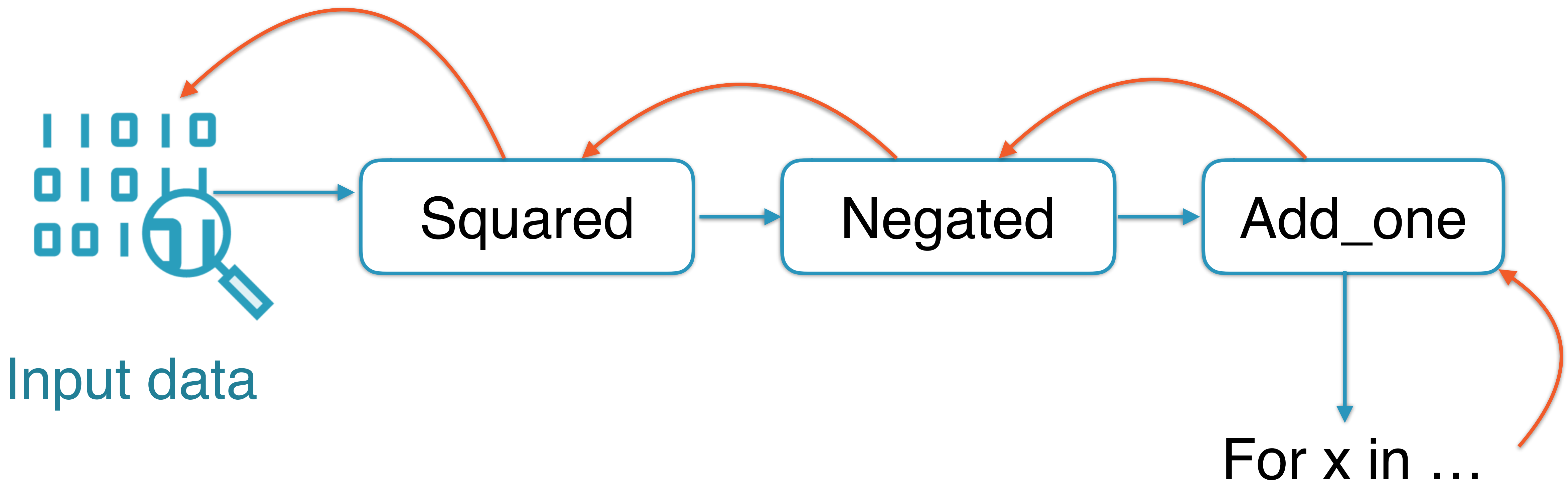
◀ **# This makes sense on its own!**

◀ **# Same here! We just negate an iterable (Given it supports __add__)**

◀ **# Same here!**

◀ **# We just need to chain the generators together to return another generator**

A Pipeline Example



Generator Based Pipeline: A Review

- 0 Storage -> it scales to infinity
- Is like unix pipelines, we can chain generators together
- To learn more -> http://www.dabeaz.com/generators/Generators.pdf

Generators can be chained to create pipelines of execution

They decouple the definition of a task from its execution

Why I Don’t Have to Remember All This: Itertools!

A Test For Itertools

```
In [2]: sample = [5,4,2,8,7,6,3,0,9,1]

In [3]: list(averager(sample))
Out[3]:
[5.0,
 4.5,
 3.6666666666666665,
 4.75,
 5.2,
 5.333333333333333,
 5.0,
 4.375,
 4.888888888888889,
 4.5]
```

- **# Given a sample ...**
- **# We print the average up to a point**
- **# The average of 5 is 5**
- **# The average of 5 and 4 is 4.5**
- **The average of 5,4,2 and 8 is 4.75 and so on...**

The Old Way

```
def old_style_averager(iterable):
    total_sum = 0
    total_elements = 0
    average = []
    for number in iterable:
        total_sum += number
        total_elements += 1
        average.append(total_sum/
total_elements)
    return average
```

- **# Given a sample ...**
- **# We iterate over it**
- **# Keep the updated average**

The Old Way

```
In [10]: sample =
np.random.random(10000)

In [11]: %timeit
old_style_averager(sample)
3.33 ms ± 63 µs per loop (mean ± std. dev.
of 7 runs, 100 loops each)

In [12]: %load_ext memory_profiler

In [13]: %memit
old_style_averager(sample)
peak memory: 71.22 MiB, increment: 0.20
MiB
```

◀ **# Not bad!**

◀ **# Keep a cap on RAM too**

The Generator Way

```python
def generator_averager(iterable):
    total_sum = 0
    total_elements = 0
    for number in iterable:
        total_sum += number
        total_elements += 1
        yield total_sum/total_elements
```

- **# Given a sample ...**
- **# We iterate over it**
- **# But we yield at each point!**

If we need a middle average, we don't iterate over all

The Generator Way

```
In [27]: sample =
np.random.random(10000)

In [28]: %timeit
generator_averager(sample)
214 ns ± 2.31 ns per loop (mean ± std. dev.
of 7 runs, 1000000 loops each)

In [29]: %memit
generator_averager(sample)
peak memory: 73.80 MiB, increment:
-0.08 MiB
```

◀ **# 94% Faster!**

◀ **# Same memory usage!**

Itertools Magic

```
In [24]: for i in itertools.starmap(pow,
[(2,5), (3,2), (10,3)]):
    ...:     print(i)
    ...:
32
9
1000
```

- **Applies the function to every tuple in the iterable!**
- **So 1000 = pow(10,3)**
- **We could apply the division if we could get a tuple that accumulates the sum and number of elements...**

Itertools Magic

```
In [26]: for i in
itertools.accumulate([2,3,4,5,6], lambda
a,b: a*b):
    ...:     print(f'-->{i}')
    ...:
    ...:
-->2
-->6
-->24
-->120
-->720
```

- Applies the function to the first element, pairing with 1
- Then it applies the result of that (2) to the next (3)
- Then it applies the result of that (6) to the next element (4)
- We could get the sums and total number of elements if we apply to tuples…

Itertools Magic

```
In [28]: for i in
enumerate(itertools.accumulate([2,3,4,5,6
]), 1):
    ...:     print(f'-->{i}')
    ...:
    ...:
-->(1, 2)
-->(2, 5)
-->(3, 9)
-->(4, 14)
-->(5, 20)
```

- **Enumerate counts the number of elements by 1**
- **So each of these tuples, if we divide them, are the average up to that point!**

Itertools Magic

```
In [29]: for i in itertools.starmap(lambda
a,b: b/a,
enumerate(itertools.accumulate([2,3,4,5,6
]), 1)):
    ...:     print(f'-->{i}')
    ...:
    ...:
-->2.0
-->2.5
-->3.0
-->3.5
-->4.0
```

```
def averager(iterable):
   return itertools.starmap(lambda a, b: b / a,
enumerate(itertools.accumulate(iterable), 1))
```

- **Combining everything we get the expected result**
- **We got our average as 1 liner!**

Itertools Magic

```
In [27]: sample =
np.random.random(10000)

In [28]: %timeit averager(sample)
 189 ns ± 1.12 ns per loop (mean ± std. dev.
of 7 runs, 1000000 loops each)

In [29]: %memit averager(sample)
peak memory: 43.12 MiB, increment: -0.08
MiB
```

◀ **# 12% Faster than generator solution**

◀ **# Half the memory usage!**

Itertools offer generator factories

We can chain them to get pretty complex generators as one liners

They are optimized in C, so they are:

- Highly scalable
- Minimal memory requirements
- Recommended all times!

Demo

Create a generator based pipeline to parse logs

Migrate much as possible to itertools to learn more on the module

Summary

Generators can be chained as pipeline to act deferred on a whole dataset

Itertools offer a lot of precooked generator factories

They can chained as a pipeline to generate pretty amazing results