Advanced Generators and Coroutines

MUCH MORE THAN JUST ITERATION: GENERATORS!

Axel Sirota
MACHINE LEARNING ENGINEER
@AxelSirota

Just Give Me the Next Item: Iteration

Iteration Examples

```
>>> for x in [1,2,3,4]:
...     print(x)
...
1
2
3
4
```

```
>>> for x in {'key1':1, 'key2':2, 'key3':3}:
...     print(x)
...
key1
key2
key3
>>>
```

```
>>> for x in open('example.txt'):
...     print(x)
...
Hello Pluralsight!

>>>
```

Special Methods

```
>>> len('Hello Pluralsight')
17
```

```
>>> min({'key1':1, 'key2':2, 'key3':
3})
'key1'
```

```
>>> max(open('example.txt'))
'Hello Pluralsight!\n'
```

```
>>> set({'key1':1, 'key2':2, 'key3':3})
{'key1':1, 'key2':2, 'key3':3}
```

For Loop Functionality

```
#  Equivalent to: for i in x: print(x) !!
>>> x = [1,2,3,4]
>>> _x = iter(x)
>>> next(_x)
1
>>> next(_x)
2
>>> next(_x)
3
>>> next(_x)
4
>>> next(_x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

◀ **# Get the iterator**

◀ **# Call next on it**

◀ **# When there are no more items it raises StopIteration**

How For Loop Works

```
_iter = iter(object)
while 1:
    try:
        x = _iter.__next__()
    except StopIteration:
        break
    # do something with x ...
```

- **# Get a copy iterator**
- **# While there are more elements**
- **# Call next on it**
- **# When there are no more items it raises StopIteration**

An **iterator** is any object that you can fetch the next element from it, via the special method __next__.

An **iterable** is any object that via a special method called __iter__ it can get an iterator

Why Generators?

Lazy

Performant

Asynchronous

An Example Iterable

```
class MyBomb:

    def __init__(self, start):
        print(f'Activating the bomb and it will explode in {start} seconds')
        self.start = start

    def __iter__(self):
        return MyBombIterator(self.start)


class MyBombIterator:

    def __init__(self, count):
        self.count = count

    def __next__(self):
        if self.count <= 0:
            print('BAMM!!')
            raise StopIteration
        value = self.count
        self.count -= 1
        return value
```

- # The Iterable creates a new Iterator
- # When there are no more items it raises StopIteration
- Each iteration the count reduces

An Example Iterable

```
>> import module2
>>> for i in module2.MyBomb(6):
...     print(i)
...
Activating the bomb and it will explode in 6 seconds
6
5
4
3
2
1
BAMM!!
>>>
```

A Different Approach: Generators as Iterators

Creating Generators

```
>>> def generator():
...     yield 'I am a generator'
...     yield 'And I count'
...     yield 1
...     yield 2
...     print('I am thinking the next one!')
...     yield 3
...
>>> mygen  =  generator()
>>> mygen
<generator object generator at
0x101735678>
```

- We can yield any number of elements
- #Any code between yields get executed
- We call the generator as a function
- # But we get a generator object, that is an Iterator

Creating Generators

```
>>> next(mygen)
'I am a generator'

>>> import inspect
>>> inspect.getgeneratorstate(mygen)
'GEN_SUSPENDED'
```

- **#When calling next we get the value**
- **And the generator gets SUSPENDED**

Creating Generators

```
>>> next(mygen)
'And I count'
>>> next(mygen)
1
>>> next(mygen)
2
>>> next(mygen)
I am thinking the next one!
3
>>> next(mygen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> inspect.getgeneratorstate(mygen)
'GEN_CLOSED'
```

- **Calling next keeps getting the next values**
- **#Notice the code in between got executed along the yield**
- **When there are no more elements It raises StopIteration**
- **And the generator is CLOSED**

Debugging Generators

```
>>> def generator():
...     print('Entering')
...     yield 1
...     print('Wait me please')
...     yield 2
...     print('I am thinking the next one!')
...     yield 3
...     print('Exiting')
...
>>> for i in generator():
...     print(f'---> {i}')
...
Entering
---> 1
Wait me please
---> 2
I am thinking the next one!
---> 3
Exiting
>>>
```

- **# We can iterate the generator with a for loop**
- **# Notice that it prints Entering and then the next print is from the caller**
- **# Each time before yielding we execute the code between yields**

Recreating Bomb Iterator

```python
def mybomb(count):
    print(f'Activating the bomb and it will explode in {count} seconds')
    while count > 0:
        yield count
        count -= 1
    print('BAM!!')
```

```
>>> import module2
>>> for i in module2.mybomb(6):
...     print(f'---> {i}')
...
Activating the bomb and it will explode in 6 seconds
---> 6
---> 5
---> 4
---> 3
---> 2
---> 1
BAM!!
>>>
```

When Performance Matters: Laziness vs. Eagerness

A Lazy Example

```
class MyNotLazyBomb:

    def __init__(self, number):
        self.number = number

    def __iter__(self):
        return MyNotLazyBombIterator(self.number)


class MyNotLazyBombIterator:

    def __init__(self, number):
        self.number = number
        self.squares = [x ** 2 for x in range(number)]
self.index = 0

    def __next__(self):
        if self.index >= len(self.squares):
            raise StopIteration
        value = self.squares[self.index]
        self.index += 1
        return value
```

◀ **# To make iteration fast we use RAM**

◀ **# We return the indexed element**

A Lazy Example

```
>>> import module2
>>> for i in module2.MyNotLazyBomb(5):
...     print(f'--->{i}')
...
--->0
--->1
--->4
--->9
--->16
```

- We return the squares now
- # The output is the same as before

A Lazy Example

```
def mylazygenerator(number):
    index = 0
    while index < number:
        yield index**2
        index += 1
```

- **# We just need to yield the square of the state**

A Lazy Example: Performance Contest

```
In [4]: %timeit for _ in module2.MyNotLazyBomb(10000): True
6.92 ms ± 93.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [5]: %timeit for _ in module2.mylazygenerator(10000): True
3.27 ms ± 73.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [6]: %memit for _ in module2.MyNotLazyBomb(10000): True
peak memory: 60.93 MiB, increment: 0.08 MiB

In [7]: %memit for _ in module2.mylazygenerator(10000): True
peak memory: 60.78 MiB, increment: -0.15 MiB

In [8]: %memit for _ in module2.MyNotLazyBomb(10000000): True
peak memory: 456.12 MiB, increment: 395.32 MiB

In [9]: %memit for _ in module2.mylazygenerator(10000000): True
peak memory: 71.82 MiB, increment: 0.00 MiB
```

Using Generator Expressions

```
for i in s:
    if condition:
        yield expression
```

```
In [21]: for i in module2.mylazygenerator(5):
    ...:     print(f'--->{i}')
    ...:
--->0
--->1
--->4
--->9
--->16
```

```
(expression for i in s if condition)
```

```
In [22]: for i in (x**2 for x in range(5)):
    ...:     print(f'--->{i}')
    ...:
--->0
--->1
--->4
--->9
--->16
```

A Lazy Example:
A Better Solution

```
class MyNotLazyBomb:

    def __init__(self, number):
        self.number = number

    def __iter__(self):
        return MyNotLazyBombIterator(self.number)


class MyNotLazyBombIterator:

    def __init__(self, number):
        self.number = number
        self.squares = {'x': x**2 for x in
range(number)}
        self.index = 0

    def __next__(self):
        try:
            value = self.squares[f'{self.index}']
        except KeyError:
            raise StopIteration
        self.index += 1
        return value
```

- **# Notice the dictionary, this is highly optimized!**
- **# Indexing in dicts is highly scalable**

A Lazy Example: A Better Solution

```
In [14]: %timeit for _ in module2.MyNotLazyBomb(10000000): True
7.22 s ± 263 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [15]: %timeit for _ in module2.mylazygenerator(10000000): True
3.25 s ± 50.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [16]: %timeit for _ in module2.MyNewNotLazyBomb(10000000): True
2.77 s ± 53.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [17]: %memit for _ in module2.mylazygenerator(10000000): True
peak memory: 72.37 MiB, increment: 0.00 MiB

In [18]: %memit for _ in module2.MyNotLazyBomb(10000000): True
peak memory: 457.94 MiB, increment: 385.62 MiB

In [19]: %memit for _ in module2.MyNewNotLazyBomb(10000000): True
peak memory: 73.36 MiB, increment: -0.01 MiB
```

Generators are great for lazy implementations

Optimizing memory for common Iterators can get clumsy

Generator expressions make the code even simpler to even 1 line of code

Demo

Create our first generator to iterate over words in a text

Analyse different implementations and performance

Summary

Generators offer a way of suspending the execution of a function

Generators yield each value every time we call next onto it

Generators can decouple the definition of an iteration from an execution