How to write a generator class?

Asked 5 years, 9 months ago Modified 1 year, 3 months ago Viewed 64k times

73

I see lot of examples of generator functions, but I want to know how to write generators for classes. Lets say, I wanted to write Fibonacci series as a class.

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1

    def __next__(self):
        yield self.a
        self.a, self.b = self.b, self.a+self.b

f = Fib()

for i in range(3):
    print(next(f))
```

Output:

```
<generator object __next__ at 0x000000000A3E4F68>
<generator object __next__ at 0x000000000A3E4F68>
<generator object __next__ at 0x000000000A3E4F68>
```

Why is the value `self.a` not getting printed? Also, how do I write `unittest` for generators?

python generator fibonacci

Share Follow

edited Mar 23, 2017 at 18:02
martineau
117k 25 161 289

asked Mar 23, 2017 at 17:56
Pritam
879 1 6 16

It's not easy to write generator class, especially for Python (If you mean by generator the generator -- docs.python.org/3/tutorial/classes.html#generators) – Konstantin Burlachenko Jul 14, 2020 at 6:47

If you mean implement iteratable protocol then it can be done (docs.python.org/3/tutorial/classes.html#iterators) and this is about what your code snippet is. – Konstantin Burlachenko Jul 14, 2020 at 6:48

Highest score (default)

106

How to write a generator class?

You're almost there, writing an *Iterator* class (I show a Generator at the end of the answer), but `__next__` gets called every time you call the object with `next`, returning a generator object. Instead, to make your code work with the least changes, and the fewest lines of code, use `__iter__`, which makes your class instantiate an *iterable* (which isn't technically a *generator*):

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1
    def __iter__(self):
        while True:
            yield self.a
            self.a, self.b = self.b, self.a+self.b
```

When we pass an iterable to `iter()`, it gives us an *iterator*:

```
>>> f = iter(Fib())
>>> for i in range(3):
...     print(next(f))
...
0
1
1
```

To make the class itself an *iterator*, it does require a `__next__`:

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1
    def __next__(self):
        return_value = self.a
        self.a, self.b = self.b, self.a+self.b
        return return_value
    def __iter__(self):
        return self
```

And now, since `iter` just returns the instance itself, we don't need to call it:

```
>>> f = Fib()
>>> for i in range(3):
...     print(next(f))
...
0
```

Why is the value self.a not getting printed?

Here's your original code with my comments:

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1

    def __next__(self):
        yield self.a          # yield makes .__next__() return a generator!
        self.a, self.b = self.b, self.a+self.b

f = Fib()

for i in range(3):
    print(next(f))
```

So every time you called `next(f)` you got the generator object that `__next__` returns:

```
<generator object __next__ at 0x000000000A3E4F68>
<generator object __next__ at 0x000000000A3E4F68>
<generator object __next__ at 0x000000000A3E4F68>
```

Also, how do I write unittest for generators?

You still need to implement a send and throw method for a `Generator`

```
from collections.abc import Iterator, Generator
import unittest

class Test(unittest.TestCase):
    def test_Fib(self):
        f = Fib()
        self.assertEqual(next(f), 0)
        self.assertEqual(next(f), 1)
        self.assertEqual(next(f), 1)
        self.assertEqual(next(f), 2) #etc...
    def test_Fib_is_iterator(self):
        f = Fib()
        self.assertIsInstance(f, Iterator)
    def test_Fib_is_generator(self):
        f = Fib()
        self.assertIsInstance(f, Generator)
```

And now:

```
>>> unittest.main(exit=False)
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 7, in test_Fib_is_generator
AssertionError: <__main__.Fib object at 0x00000000031A6320> is not an instance
of <class 'collections.abc.Generator'>

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
<unittest.main.TestProgram object at 0x0000000002CAC780>
```

So let's implement a generator object, and leverage the `Generator` abstract base class from the collections module (see the source for its implementation), which means we only need to implement `send` and `throw` - giving us `close`, `__iter__` (returns self), and `__next__` (same as `.send(None)`) for free (see the Python data model on coroutines):

```
class Fib(Generator):
    def __init__(self):
        self.a, self.b = 0, 1
    def send(self, ignored_arg):
        return_value = self.a
        self.a, self.b = self.b, self.a+self.b
        return return_value
    def throw(self, type=None, value=None, traceback=None):
        raise StopIteration
```

and using the same tests above:

```
>>> unittest.main(exit=False)
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
<unittest.main.TestProgram object at 0x00000000031F7CC0>
```

Python 2

The ABC `Generator` is only in Python 3. To do this without `Generator`, we need to write at least `close`, `__iter__`, and `__next__` in addition to the methods we defined above.

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1
    def send(self, ignored_arg):
        return_value = self.a
        self.a, self.b = self.b, self.a+self.b
```

```
        return self
    def next(self):
        return self.send(None)
    def close(self):
        """Raise GeneratorExit inside generator.
        """
        try:
            self.throw(GeneratorExit)
        except (GeneratorExit, StopIteration):
            pass
        else:
            raise RuntimeError("generator ignored GeneratorExit")
```

Note that I copied `close` directly from the Python 3 standard library, without modification.

Share Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 1

answered Mar 23, 2017 at 18:07
Russia Must Remove Putin ♦
359k 86 397 330

Hi Aaron, thank you so much for your response, this is exactly what I was looking for. What is the best way to learn more about iterators and generators ? – Pritam Mar 23, 2017 at 18:44

@Pritam I expand a lot on that subject in this answer here: stackoverflow.com/a/31042491/541136 – Russia Must Remove Putin ♦ Mar 23, 2017 at 19:51

@AaronHall In the second instantiation `f = iter(Fib())` , (after "And now:"), you probably meant to instantiate without wrapping the Fib class in the `iter` function? – loxosceles Jan 3, 2020 at 20:54

1 @loxosceles The intention is to demonstrate the usage of the iterator protocol. An iterator has an `__iter__` method that, when called, returns itself. It might seem redundant, but it's what's called when the iterator object is placed in an iteration context (like a for loop or passed to an iterable constructor). – Russia Must Remove Putin ♦ Jan 3, 2020 at 21:23

1 Your reply to "How to write a generator class?" only explains how to implement the iterator interface. Your response is organized incoherently. By the end you show how to copy the implementation for a coroutine from the cpython source code... That has nothing to do with implementing the generator interface in a class. Copying undocumented implementation code from cpython's source is bad practice, because it can break between minor releases. Not only that, but since it is not part of any PEP spec it may only work with cpython and break on other interpreters. – Jared Deckard Feb 17, 2020 at 15:40

8

`__next__` should *return* an item, not yield it.

You can either write the following, in which `Fib.__iter__` returns a suitable iterator:

```
class Fib:
    def __init__(self, n):
        self.n = n
        self.a, self.b = 0, 1
```

```
            yield self.a
            self.a, self.b = self.b, self.a+self.b

f = Fib(10)

for i in f:
    print i
```

or make each instance itself an iterator by defining `__next__`.

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        x = self.a
        self.a, self.b = self.b, self.a + self.b
        return x

f = Fib()

for i in range(10):
    print next(f)
```

Share Follow

answered Mar 23, 2017 at 18:06

chepner
473k 69 497 645

4

If you give the class an `__iter__()` method implemented as a generator, "it will automatically return an iterator object (technically, a generator object)" when called, so *that* object's `__iter__()` and `__next__()` methods will be the ones used.

Here's what I mean:

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1

    def __iter__(self):
        while True:
            value, self.a, self.b = self.a, self.b, self.a+self.b
            yield value

f = Fib()

for i, value in enumerate(f, 1):
    print(value)
```

Output:

```
0
1
1
2
3
5
```

Share Follow

edited Sep 20, 2021 at 10:55

answered Mar 23, 2017 at 18:13

martineau
117k 25 161 289

1 This makes it an iterable, not a generator – Brian M. Sheldon Jul 25, 2017 at 14:57

@Brian: Better? – martineau Jul 25, 2017 at 17:31

Yes this makes it a proper generator class – Brian M. Sheldon Jul 28, 2017 at 12:51

3

Do not use `yield` in `__next__` function and implement `next` also for compatibility with python2.7+

Code

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1
    def __next__(self):
        a = self.a
        self.a, self.b = self.b, self.a+self.b
        return a
    def next(self):
        return self.__next__()
```

Share Follow

edited Mar 23, 2017 at 18:09

Willem Van Onsem
422k 29 395 516

answered Mar 23, 2017 at 18:08

Sarath Sadasivan Pillai
6,452 28 42

3

Using `yield` in a method makes that method a **generator**, and calling that method returns a **generator iterator**. `next()` expects a generator iterator which implements `__next__()` and `return` s an item. That is why `yield` ing in `__next__()` causes your generator class to output generator iterators when `next()` is called on it.

https://docs.python.org/3/glossary.html#term-generator

iterator.

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1
        self.generator_iterator = self.generator()

    def __next__(self):
        return next(self.generator_iterator)

    def generator(self):
        while True:
            yield self.a
            self.a, self.b = self.b, self.a+self.b

f = Fib()

for i in range(3):
    print(next(f))
# 0
# 1
# 1
```

Share Follow

answered Feb 14, 2020 at 15:56

Jared Deckard
593 5 15

Citing the glossary is usually a good thing, but in this case I believe it is imprecise to the point of being incorrect. See my answer for my reasoning. Where the glossary is contradicted by the implementation, the implementation would be the correct source. – Russia Must Remove Putin ♦ Feb 14, 2020 at 16:55

2 Differentiating "generator" and "generator iterator" is an important part of the answer to this question. The glossary is the most precise source available. The glossary does not contradict the implementation. You are conflating iterators and coroutines with generators, but they are not the same. – Jared Deckard Feb 17, 2020 at 15:36