Could we implement generator using class? [duplicate]

Asked 5 years, 4 months ago Modified 5 years, 4 months ago Viewed 1k times

-1

This question already has answers here:

How to write a generator class? (5 answers)

Closed 5 years ago.

when we look at the Python documentation we could see that generators are always defined using yield statement, but in the Internet we could see that some people are trying to implement generators using classes (eg. here How to write a generator class?).

Here is example generator implementation using classes:

```
from collections import Generator
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
```

When we execute it in repl we can see it is not the generator, but ordinary object. It only tries to behave like generator.

```
>>> x = Fib()
>>> x
<__main__.Fib object at 0x7f05a61eab70>
```

When we look at PEP 342:

4. Add a close() method for generator-iterators, which raises GeneratorExit at the point where the generator was paused.

Share Improve this question Follow

edited Sep 9, 2017 at 17:46

Martijn Pieters ♦
1.0m 282 3947 3283

asked Sep 9, 2017 at 17:41

Adam
629 9 21

It's not acting like a generator. You printed an object. stackoverflow.com/questions/4932438/… – OneCricketeer Sep 9, 2017 at 17:47

1 Answer

Sorted by: Highest score (default)

3

Generators are simply a type of *iterator*. From the datamodel documentation:

> Generator functions
> A function or method which uses the `yield` statement [...] is called a generator function. Such a function, when called, **always returns an iterator object** which can be used to execute the body of the function: calling the iterator's `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a return statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

You can't tell by the `repr()` output if something is a generator. Python looks for the iterator methods, and you can implement your own `send` and `throw` methods on top of those, as you have done.

As such, your implementation *works as designed*, it is valid iterator:

```
>>> x = Fib()
>>> next(x)
0
>>> next(x)
1
```

Without the `collections.abc.Generator` base, you can also implement your own `__iter__` method (this *has* to return `self`), and a `__next__` method that produces the next value when

Martijn Pieters ♦
1.0m 282 3947 3283

The question is a using coroutine object – OneCricketeer Sep 9, 2017 at 17:50

@cricket_007: no it is not. – Martijn Pieters ♦ Sep 9, 2017 at 17:52

1 @cricket_007: *Coroutines **also** have the methods listed below, which are **analogous to those of generators***. – Martijn Pieters ♦ Sep 9, 2017 at 17:54

2 @Adam: that's a philosophical question. Is an object that has all the same methods as a list, really a list? Python *doesn't care*; as long as other code can use it *just like you can use a generator*, it's all fine. No, it's not a generator, but it walks and talks like one. – Martijn Pieters ♦ Sep 10, 2017 at 14:37

1 @Adam: the `collections.abc` objects have two functions: letting you create objects that walk and talk like those types, *and* testing if for those types. If your goal is to produce an object that can stand in for a generator, then using `collections.abc.Generator` as a base is a great way of doing that. – Martijn Pieters ♦ Sep 10, 2017 at 14:39