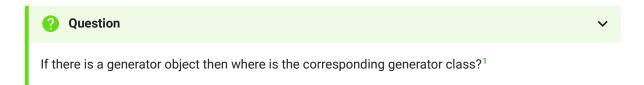
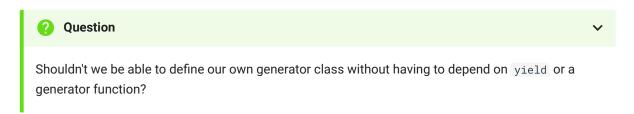
Generator Class

We saw that calling a generator function returns a generator object.



We also saw that next is not that special and driving a generator does not require the special keyword yield.



Minimal example from scratch

Such curiosity would help us demystify the implementation of generators. Let's try to write a minimal class, from scratch, to replace the generator function my_simple_range .

Generator Class

```
class MySimpleRangeMinimal:
    def __init__(self, start: int, stop: int):
        self.start = start
        self.stop = stop
        self.i = self.start

def send(self, value):
        if self.i < self.stop:
            out = self.i
            self.i = self.i + 1
            return out
        raise StopIteration</pre>
```

```
def __next__(self):
    return self.send(None)

def __iter__(self):
    return self

print(list(MySimpleRangeMinimal(0, 6)))
# [0, 1, 2, 3, 4, 5]
```

Generator Function

```
Python

def my_simple_range(start: int, stop:int):
    i = start
    while i < stop:
        yield i
        i = i + 1

print(list(my_simple_range(0, 6)))
# [0, 1, 2, 3, 4, 5]</pre>
```

The class MySimpleRangeMinimal performs the same basic task as the generator function my_simple_range without needing to use yield or any fancy concepts such as simple or extended functions and explicit or implicit control transfer suspendables. Not only that, calling the generator function using $my_simple_range(0, 6)$ is early similar to calling the constructor of the class in MySimpleRangeMinimal(0, 6). We hinted at this earlier in the course.

The class MySimpleRangeMinimal does not have any dependency but it also has a few drawbacks. Firstly, it does not implement throw and close, which are the other methods required to a generator. Secondly, it appears to have boilerplate code in __next__ and __iter__ if we compare it to the source. Thirdly, it's not a generator even if it can perform the same basic task as a generator. You can see it yourself as shown below.

```
Python

print(type(my_simple_range(0, 6)))
# <class 'generator'>

print(type(MySimpleRangeMinimal(0, 6)))
# <class '__main__.MySimpleRangeMinimal'>

from collections.abc import Generator
issubclass(MySimpleRangeMinimal, Generator)
# False
```

Improved example using ABC

We can improve upon all of the shortcomings by simply inheriting from collections.abc.Generator. It helps to look at the source for Generator, as we're writing the following code.

```
Python
from collections.abc import Generator
class MySimpleRange(Generator):
    def __init__(self, start: int, stop: int):
        self.start = start
        self.stop = stop
        self.i = self.start
    def send(self, value):
        if self.i < self.stop:</pre>
            out = self.i
            self.i = self.i + 1
            return out
        raise StopIteration
    def throw(self, typ, val=None, tb=None):
        super().throw(typ, val, tb)
print(list(MySimpleRange(0, 6)))
# [0, 1, 2, 3, 4, 5]
print(type(MySimpleRange(0, 6)))
# <class '__main__.MySimpleRange'>
issubclass(MySimpleRange, Generator)
# True
```

The constructor and the send method are the same for both MySimpleRangeMinimal and MySimpleRange. But, we did not need to write __next__ and __iter__ boilerplate methods for MySimpleRange at the expense of writing throw, which simply calls the method from the super class. Since we inherited from Generator, it is expected that MySimpleRange be a subclass of Generator. It may be interesting to note that my_simple_range(0, 6) returns a generator-type object. The type generator is an built-in type. Neither MySimpleRangeMinimal(0, 6) nor MySimpleRange(0, 6) are objects of any built-in type.

More importantly, MySimpleRange provides the close and throw methods which MySimpleRangeMinimal did not. If you chose to write a generator class (instead of a generator function) in the real world, then inheriting from Generator is the better way.

```
Python

x = MySimpleRange(0, 6)
next(x) # 0
x.close()

y = MySimpleRangeMinimal(0, 6)
next(y) # 0
y.close()
```

Running cost vs fixed cost

Notice how much more verbose and complicated <code>MySimpleRange</code> is compared to <code>my_simple_range</code>. For the generator class, we needed to inherit from <code>Generator</code>, write a constructor method, write a send method that raises <code>StopIteration</code>, write a throw method even if all it does is call its super, and then link all of these things together into a working class. For the generator function, we were able to avoid all of these chores in lieu of the one-time expense of having to define and learn about suspendable functions. Some would argue that the reduced running cost of writing generator functions is worth the higher, cognitive, initial fixed cost of learning suspendable functions.

Footnotes

1. Quick answer: if a generator object is created by calling a generator function, then the corresponding class is only implicitly defined. ←

Last update: 2022-09-13 Created: 2022-09-13

Comments

0 reactions



0 comments

