teclado

ADVANCED COLLECTIONS

Day 23: Generators and Generator Expressions

Welcome to day 23 of the 30 Days of Python series! Today we're going to be looking at some ways of creating our own iterators using generators and generator expressions.

We're also going to be looking at an important function called `iter` which returns an iterator for any iterable we pass to it. This is going to let us confirm a lot of the theory we discussed in yesterday's post, and we're also going to be able to use it to get a deeper understanding of `for` loops.

This is post is going to build a great deal on what we discussed in

yesterday's post, so if you haven't read it yet, I'd recommend you take a look before reading any further.

The `iter` function

Python has a built in function called `iter` which returns an iterator for the iterable we provide as an argument.

For example, let's take a simple list of numbers like this:

```
numbers = [1, 2, 3, 4, 5]
```

If we pass this list of numbers to `iter` we'll get back an iterator for that list.

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)

print(numbers_iter)  # <list_iterator object at 0x7f57d138af7
0>
```

In this case we get a `list_iterator` object, which is going to let us access values in `numbers` . Different types have their own iterators which understand how to give us items from those iterables. Getting elements from a dictionary is somewhat different from getting items from a list, after all.

We can use this `list_iterator` object just like any other iterator. We can pass it to `next` , for example.

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)

print(next(numbers_iter))  # 1
print(next(numbers_iter))  # 2
```

One interesting question is, what happens when we call `iter` on the `list_iterator` ?

This is perfectly legal, because `iter` just expects an iterable, and all iterators are iterables. It also produces an interesting effect.

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)

print(numbers_iter is iter(numbers_iter))  # True
```

We find that passing `numbers_iter` to the `iter` function causes `iter` to return the very same iterator. This might seem odd at first, but it makes quite a lot of sense.

Yesterday we talked about iterators being the means by which we access items in an iterable. When we want to iterate over an iterable, we need to ask for an iterator that knows how to get those values.

If we ask the *iterator* to give us a way to access those values, it offers up itself, since it's already capable of doing what we want.

Replicating `for` loops with `iter`

One cool thing we can do with the `iter` function is replicate the behaviour of Python's `for` loop. This is going to give us a little peek at what `for` loops really do behind the scenes.

Let's use our list of numbers for this example again.

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)
```

We have an iterator, which is an important first step, but we need a couple of other tools to make this work. First, we need a `while` loop, because we want to loop a potentially infinite number of times. Second, we need a `try` statement so that we can look out for a `StopIteration` exception.

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)

while True:
        try:
                number = next(numbers_iter)
        except StopIteration:
                break
        else:
                print(number)
```

And just like that we have a `for` loop written with `while` .

We have our loop variable `number` defined inside the `try` , and we have the loop body inside the `else` clause. Once we run out of numbers, the loop is going to terminate, just like we see with a `for` loop.

This is actually extremely close to how an actual `for` loop works under the hood. It does request an iterator for whatever we want to iterate over, and it does call `next` to retrieve values from that iterator. When a `StopIteration` is raised, Python handles that error by breaking the loop.

This isn't something you should be doing in your production code, but it's an interesting peek behind the curtain that helps us better understand the structures we've been using since week 1.

Generators

Let's leave `iter` alone for a moment and turn to the topic of creating

Let's leave `iter` alone for a moment and turn to the topic of creating our own iterators using *generators*.

There are quite a few ways to create custom iterators in Python, but most of them are beyond the scope of this series. This isn't really much of a limitation though, and we can do a great deal of very complicated things using generators.

The generator syntax is actually going to be very familiar to us, because a generator is actually just a function. The only thing which differentiates a generator from a regular function is a special keyword called `yield`.

Before we dive into this new `yield` keyword, let's look at a simple generator example.

```
def first_hundred():
        for number in range(1, 101):
                yield number
```

Here I've defined a generator, which is just a special function, and I've called it `first_hundred`.

We can see from the function body that it has something to do with the numbers `1` to `100` inclusive, and we can probably infer that it's going to give us the first hundred integers, starting with `1`.

Let's call our function and see what happens.

```
def first_hundred():
        for number in range(1, 101):
                yield number

g = first_hundred()
print(g)
```

If you run this code, we certainly don't get anything like the numbers `1` to `100` printed to the console. We get this `generator` object:

```
<generator object first_hundred at 0x7faaa563fc80>
```

This is actually called a *generator iterator*, which is what gets returned when we call any function that contains the `yield` keyword.

As the name would imply, this is an iterator, and we can use it just like any other.

```
def first_hundred():
        for number in range(1, 101):
                yield number

g = first_hundred()

print(next(g))  # 1
print(next(g))  # 2
print(next(g))  # 3
```

Important

When we call a generator, it gives us back a new generator iterator. Each of these generator iterators is an independent iterator, so be careful you don't do something like this:

```
def first_hundred():
        for number in range(1, 101):
                yield number

print(next(first_hundred()))  # 1
print(next(first_hundred()))  # 1
print(next(first_hundred()))  # 1
```

Each call to `first_hundred` gave us a new iterator, so we're only getting the first value from each one. You also don't assign the iterator

getting the first value from each one. You also don't assign the iterator

anywhere, so it's not really possible for us to call `next` on the same iterator again.

The `yield` keyword

Now that we've seen a generator in action, it's time to talk about what this `yield` keyword is doing.

We know already that it signals to Python that we're defining a generator, but it also seems to have some role in actually providing the values we want from the resulting generator iterator.

What `yield` actually does is create a pause in the execution of the function body. When we call `next` and pass in our generator iterator, the code in the function body is going to run until we hit that `yield` keyword.

The value after the `yield` keyword is what we actually want to provide before we pause the execution of the function body. In this way we can think of `yield` as something like a non-terminating `return` statement.

We can see all this by adding a few `print` calls to `first_hundred`.

```
def first_hundred():
        print("First value requested\n")

        for number in range(1, 101):
                print("Starting new iteration")
                yield number
                print("Ending this iteration\n")

g = first_hundred()
```

At this point, nothing is printed. The generator iterator has been created, but we haven't actually tried to access any values. Now let's

created, but we haven't actually tried to access any values. Now let's pass `g` to `next` a couple of times.

```
def first_hundred():
        print("First value requested\n")

        for number in range(1, 101):
                print("Starting new iteration")
                yield number
                print("Ending this iteration\n")

g = first_hundred()

print(next(g))
print(next(g))
```

Now our output looks like this:

```
First value requested

Starting new iteration
1
Ending this iteration

Starting new iteration
2
```

First we get the `"First value requested\n"` string, and then we enter the `for` loop. At this point we get a value from the `range` object, which is assumed to `number`, and we print the `"Starting new iteration"` string.

We then encounter the `yield` keyword which pauses the execution of the function body, and our generator iterator spits out `1`, which is the current value of `number`. This value is returned by the call to `next` and we print it to the console.

We then call `next` again, and we continue from where we left off. This

We then call `next` again, and we continue from where we left off. This means we print the `"Ending this iteration\n"` string, and we move onto a new iteration of the `for` loop.

We call the `print` function at the start of the loop again, and then we hit the `yield` one more time. We yield the number, which is what `next` returns once again. This is then printed to the console, just as before.

For this second iteration, you'll note that we don't print the `"Ending this iteration\n"` string, because `yield` paused the execution before we reached that point.

If we were to call `next` again, we'd get this string printed first, before starting a third iteration of the loop.

Note

`yield` is actually a very complicated keyword, and it can do a great deal more than what we're using it for. We're not going to be covering this additional behaviour in this series, however, because it only has applications in much more advanced code.

I'm mentioning this only so that you know there is more to learn once you're a little further along in your Python career.

Generator Expressions

In addition to creating generator iterators through functions, we can also use *generator expressions*.

The generator expression syntax is also going to be very familiar to us, because it's exactly the same as the comprehension syntax we say in day 15. The only difference is that we use regular parentheses, rather than square brackets or curly braces.

We can use them very much like comprehensions, but they come with

all the benefits of iterators that `map` and `filter` provide. If you've wanted to have those benefits, but didn't like the syntax for `map` and `filter`, generator expressions are for you.

For example, let's create a simple generator expression that squares every number in a `range`.

```
squares = (number ** 2 for number in range(1, 11))
```

Since `squares` refers to an iterator, printing it directly doesn't give us anything too useful, but it does at least confirm we're working with a generator iterator.

```
<generator object <genexpr> at 0x7f33225a0c80>
```

If we want to get values out, we can either pass it to a `for` loop, we can destructure it, or we can use `next` to perform manual iteration.

```
squares = (number ** 2 for number in range(1, 11))

for square in squares:
        print(square)

squares = (number ** 2 for number in range(1, 11))

print(*squares, sep=", ")

squares = (number ** 2 for number in range(1, 11))

print(next(squares))  # 1
print(next(squares))  # 4
print(next(squares))  # 9
```

Remember that the values in `squares` get consumed when we iterate

over the iterator, so you need to redefine `squares` if you want to iterate over it more than once.

Style note

One nice thing about generator expressions is that we can forego the parentheses when we use the generator expression as the sole argument in a function or method.

This is totally legal syntax for example:

```
total = sum(number ** 2  for number in  range(1,  11))
print(total)  # 385
```

This helps us reduce nested brackets when they would only hinder readability.

Exercises

1) Write a generator that generates prime numbers in a specified range. You can make use of your solution to exercise 3 from day 8 as a starting point.

2) Below we have an example where `map` is being used to process names in a list. Rewrite this code using a generator expression.

```
names = [" rick", " MORTY  ", "beth ", "Summer", "jerRy    "]
names = map(lambda name: name.strip().title(), names)
```

3) Write a small program to deal cards for a game of Texas Hold'em. The order of the deal is as follows:

- The deck is shuffled.
- One card is handed to each player in order.
- A second card is handed to each player order.

Then comes the more complicated part of the deal.

- First, the top card of the deck is discarded. This is called the burn.
- Three cards are then placed in the centre of the table, which is called the *flop*.
- Another card is burned, meaning we discard another card from the top of the deck.
- We add another card to the centre, which is called the *turn*.
- We burn another card.
- Finally, there's the *river*, where a fifth and final card is added to the centre.

The desired output for the program is something like this:

```
How many players are there? 2

Player 1 was dealt: (4, hearts), (4, clubs)
Player 2 was dealt: (9, clubs), (jack, diamonds)

The flop: (jack, clubs), (4, diamonds), (king, spades)
The turn: (8, hearts)
The river: (ace, hearts)
```

As the example would indicate, the program should accept a variable number of players. There must be at least 2 players, and no more than 10.

After the flop, the turn, and the river there's usually a round of betting, so if you want to extend this exercise, you may want to give the user the option to pause at each of these points.

Hint: We can shuffle cards using the `random.shuffle` method. This shuffles a sequence in-place, which means it modifies the original sequence. We can then create an iterator from that sequence using `iter` to make is easy for us to retrieve cards one at a time.

You can find documentation for `random.shuffle` here.

© 2021 Teclado Ltd.