teclado

ADVANCED COLLECTIONS

Day 22: Iterators

Welcome to week 3 of the 30 Days of Python series! Today we're going to be taking a look at iterators!

Though we've not encountered this term *iterator*, we've actually seen lots of iterators already. For example, any time we call `zip`, `enumerate`, `map`, or `filter`, what we get back is an iterator.

Today we're going to dive a little into what they are, and why they're important, as well as a few gotchas we need to watch out for.

What are iterators?

Iterators have a technical definition in Python, but we haven't yet

covered many of the concepts required to properly understand this

definition. Despite this, we can still get a broad overview of what iterators are through some relatively intuitive concepts.

Let's start out by talking about *iterables*, since this is a term we've used a number of times throughout the series, and it's a related term.

An iterable is really any value we can iterate over with something like a `for` loop. We can also destructure iterables, and we can create collections from them by passing the value to functions like `list` or `tuple`.

Here are some of the iterable types we've encountered so far:

- strings
- lists
- tuples
- dictionaries
- sets
- `range` objects
- `zip` objects
- `enumerate` objects
- `map` objects
- `filter` objects

If we want a very broad stroke definition, an iterable is really just a thing we can get values out of one at a time.

One question we should probably be asking ourselves is, how do we get hold of those values? How does Python know what to give us?

The answer to this question, is that it uses iterators. If iterables are the things we can get values out of, iterators are the things which give us

those values. They're the mechanism Python uses to extract values from iterables.

As we can see from our list of iterables, the *iterators* I mentioned in the introduction to this post are also *iterables*. In fact, *all* iterators are iterables.

There's a technical reason for this to do with how iterators are defined, but it also makes some intuitive sense.

When we want to get values from an iterable as part of some iteration, we're saying,

> "Give me the iterator in charge of providing your values."

We then get put in touch with that iterator, and it starts releasing values to us, one at a time.

If we ask an iterator directly, it says,

> "Actually, *I'm* in charge of releasing the values, so *I* can give you them."

Important

Not all iterators retrieve values from another iterable. Many iterators spontaneously generate values for us instead.

There are plenty of examples of this kind of iterator in the itertools module.

This works because iterators are iterables (which means we can iterate over them to get the values we want), and iterators are capable of providing values to us without the aid of another iterator. All it requires is for an iterator to have an means of internally generating values, rather than providing a means for us to access values defined elsewhere.

Important properties of iterators

Iterators have several important and useful properties that we should be aware of.

Iterators are often lazy

First of all, most iterators are lazy types, particularly those defined in the standard library. This means they don't bother calculating all of their values in advance. We've encountered this for some non-iterator types as well, like `range` objects.

Being lazy comes with some significant memory benefits, because we only have to keep the most recent value in memory. Once we're done using it for whatever it is we're doing, we can throw it away, potentially freeing up that memory.

There are also some potential savings in terms of computations as well, because it may be the case that we don't need every value that an iterator can provide. If we only care about the first 3 values, there's no need to figure out what the next 200 are.

These benefits don't come for free though: there are compromises involved.

First, it's often not possible to determine the length of a lazy type, because the number of items it contains may become apparent only after calculating those values.

For example, let's say we're trying to filter a list of words so that we only keep words starting with `"a"`. If you're not familiar with `filter` and `methodcaller`, check out day 20.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
```

```
a_words = filter(methodcaller("startswith", "a"), words)
```

It should be pretty clear that we can't determine the length of `a_words` until we actually performing the filtering operation. However, `filter` is going to wait until we ask it for values to calculate anything. It's therefore impossible for us to get any indication of how long `a_words` actually is.

In a similar vein, it's also impossible for us to readily see what *values* the `filter` object contains. In order to find out, we have to actually request the values from the iterator, and this can lead to some additional problems as we'll see in a moment.

Iterator values are consumed

This is a really important one to keep in mind, because it's a common "gotcha" that often trips up newer Python developers.

This can be easily demonstrated using our `filter` example above. Let's just use a `for` loop to iterate over the values and see what we have left after filtering.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

for word in a_words:
        print(word)
```

So far so good. We get all of the words starting with `"a"` printed to the console.

```
anaconda
anime
addition
```

```
addition
```

Now what if we add another `for` loop to do the exact same thing?

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

for word in a_words:
        print(word)

for word in a_words:
        print(word)
```

We get the exact same output.

```
anaconda
anime
addition
```

What we would usually expect here is for the names to print twice, but that's not what happens.

Let's instead try to turn `a_words` into a list after the `for` loop to see what's inside.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

for word in a_words:
        print(word)
```

```
a_words = list(a_words)
print(a_words)  # []
```

What we get is an empty list.

The reason this is happening is because there's nothing left to put into our new list after we've iterated over our `iterator`. The values have already been provided to us, and the iterator won't given them to us a second time.

This can often lead to problems if you try to see what's inside an iterator when checking your code, because by unpacking the iterator and printing its values, you're consuming the values. If you have code after the `print` call which relies on those values, you may get an unexpected exception or, at the very least, unintended behaviour.

Changes to mutable collections can affect an iterator

Let's use our filtering example once again.

This time we're going to create `a_words` from our `filter`, and then I'm going to make some modifications to the `words` list. We're then going to print `a_words` to see what effect this had, if any.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

words.append("apple")

for word in a_words:
        print(word)
```

Despite the fact we've already created the `filter` iterator before modifying `words`, the new word still gets included in the `filter`.

```
anaconda
anime
addition
apple
```

This is another side effect of laziness, and it makes a fair bit of sense. `filter` hasn't actually done any filtering when we define `a_words`, and it doesn't store its own version of the list to grab its values from. It just references the one we already have in memory.

If we modify that list, and then we ask `filter` to filter values from the list after that change, it doesn't even know that new items were added. It doesn't really care. It's just going to keep giving us values until it runs out.

Manual iteration with the `next` function

Now that we have an idea about what iterators are and we know a little about their properties, let's turn to some interesting ways to use them.

One quite useful way we can work with iterators is through manual iteration. That is, requesting values from an iterator one at a time, without the aid of tools like unpacking, or `for` loops.

The way we request a new item from an iterator is using the `next` function. When we call `next`, it expects a reference to an iterator. This will not work for regular iterables!

To start, let's try to grab the first word in our `words` list that begins with `"a"`.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

first_word = next(a_words)
print(first_word)  # "anaconda"
```

Of course we can also pass the return value of `next` to `print` directly as well.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

print(next(a_words))  # "anaconda"
```

As we can see, we got back `"anaconda"`, which is the first `"a"` word in the `words` list.

If we call `next` again, we get `"anime"`: the second `"a"` word in `words`.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

print(next(a_words))  # "anaconda"
print(next(a_words))  # "anime"
```

So, what is a practical application for this?

Remember the iris data from day 14? Originally, we used an approach like the one below to get the file data and extract the headers.

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

headers = iris_data[0].strip().split(",")
irises = []

for row in iris_data[1:]:
        iris = row.strip().split(",")
        iris_dict = dict(zip(headers, iris))

        irises.append(iris_dict)
```

This approach worked fine, but it's not the best approach if this was a very large file for a couple of reasons:

1. By using `readlines` we ended up storing the entire file in memory as a list right away.
2. By using slicing in the `for` loop we created another list in memory which contains almost all of the same data as the first one.

We can avoid all of this by making use of iterators. It turns out that the `open` function already gives us back an iterator, so we don't have to do a lot to make this work.

```
irises = []

with open("iris.csv", "r") as iris_file:
        headers = next(iris_file).strip().split(",")

        for row in iris file:
```

```
        iris = row.strip().split(",")

        iris_dict = dict(zip(headers, iris))

        irises.append(iris_dict)
```

Here we start by requesting the first line of the file from the `iris_file` iterator. We then process this string as usual and assign it to `headers`.

Remember that this consumes the value, so when we iterate over `iris_file`, we're only going to get line 2 onward.

Moreover, we only work with one line at a time, and we don't keep the line around after we're done with it. We just create the dictionary we need, and we throw the original string away in favour of a new one. This is extremely memory efficient compared to what we were doing before.

Depending on what we wanted to do with this data, there are ways to do all of our computations without ever having to create some collection to store all of the values.

The `StopIteration` exception

Manual iteration is very useful, but we have to be a little careful, because we may encounter an exception called `StopIteration`. While not exactly an error, if we don't properly handle this exception, our program will still terminate.

We spoke a little about `StopIteration` in day 19, but not in great depth. Just as a quick reminder, the `StopIteration` exception is raised when an iterable has no more values left to give us when we iterate over it.

This exception is handled for us when we use a `for` loop, or when we destructure a collection, but not when we perform manual iteration.

For example, if we look again at our `filter` example, we will get a

For example, if we look again at our "filter" example, we will get a `StopIteration` exception raise if we try to request four items.

```
from operator import methodcaller

words = ["anaconda", "peach", "gravity", "cattle", "anime",
"addition"]
a_words = filter(methodcaller("startswith", "a"), words)

print(next(a_words))  # "anaconda"
print(next(a_words))  # "anime"
print(next(a_words))  # "addition"

print(next(a_words))  # StopIteration
```

Because of this, it's a very good idea when performing manual iteration to catch any potential `StopIteration` exception using a `try` statement.

Exercises

1) Below you'll find a list containing several tuples full of numbers:

```
numbers = [(23, 3, 56), (98, 1034, 54), (254, 344, 5), (45, 2
), (122, 63, 74)]
```

2) Use the `map` function to find the sum of the numbers in each tuple. Use manual iteration to print the first two results provided by the resulting `map` object.

3) Imagine you have 3 employees and it's been agreed that the employees will take it in turns to lock up the shop at night. This means that for employees A, B, and C, employee A will close the shop on day 1, then B will close the shop on day 2, C will close the shop on day 3, and then we start the cycle again with employee A.

Write a program to create a schedule that lists which of your employees

will lock up the shop on a given day over a 30 day period. You should

list the day number, the employee name, and the day of the week. You can choose any employee to lock the shop on day 1, and you can also choose which day of the week day 1 corresponds to.

You should make use of the `cycle` function in the `itertools` module to create a repeating series of values. You can find documentation here.

Once you've finished with the exercises, you can check your solutions against ours here.
