WORKING WITH MULTIPLE FILES

Day 20: map, filter, and Conditional Comprehensions

Welcome to day 20 of the 30 Days of Python series! Today we're going to be expanding our use of comprehensions to incorporate filtering conditions. This is going to allow us to create a new collection from some subset of the original values.

We're also going to be looking at a couple of function based alternatives to the comprehension syntax called `map` and `filter`.

A quick recap of comprehensions

We've used comprehensions quite a few times now, so I don't really want to recap the syntax here. What I want to talk about is what comprehensions are *for*.

We use a comprehension when we want to make a new collection from some other iterable. However, there are cases where we want to make a new collection and a comprehension isn't necessary. We only use a comprehension when we want to change something about the values when we make this new collection. For example, we might want to turn every string in a list to title case:

```
names = ["tom", "richard", "harold"]
names = [name.title() for name in names]
```

If we just want to turn the `names` list to, say, a set, we can do this instead:

```
names = ["tom", "richard", "harold"]
names = set(names)
```

We don't have to bother with a much more verbose set comprehension:

```
names = ["tom", "richard", "harold"]
names = {name for name in names}
```

With this in mind, we can really think of comprehensions as a way of performing an action for every item in some iterable, and then storing the results.

The `map` function

Now that we have this mental model of comprehensions, the parallel to `map` is going to be very clear.

`map` is a function that allows us to call some other function on every item in an iterable. And what are functions? They're just a means of defining some action we want to perform in a reusable way. In other words, `map` is a way of performing some action for every item in an iterable, just like a comprehension.

Let's say I want to cube every number in a list of numbers. We can use `map` like so:

```
def cube(number):
        return number ** 3

numbers = [1, 2, 3, 4, 5]
cubed_numbers = map(cube, numbers)
```

One thing we should take note of here is that we've only passed in name of the function we want `map` to call for us. We haven't called that function ourselves.

In the example above, our function is called `cube`, so we pass in `cube`, not `cube()`.

Some of you may have tried to print `cubed_numbers` from the example above, but perhaps you didn't get what you expected. Instead of a list of cubed numbers, we get something like this:

```
<map object at 0x7f8a284ab3d0>
```

This is because `map` objects are another *lazy* type, like the things we get back from calling `zip`, `enumerate`, or `range`. `map` doesn't actually bother calculating any values until we ask for them.

This allows `map` to be a lot more memory efficient than the comprehensions we've looked at so far, because it doesn't have to store all the values at once. It also may never end up calculating any values at all, since it only calculates the next value when we request it. If we never request the values, they never get calculated!

So how can we get things out of a `map` object? Well, they're iterable, so we can iterate over the values:

```
def cube(number):
        return number ** 3
```

```
numbers = [1, 2, 3, 4, 5]
cubed_numbers = map(cube, numbers)

for number in cubed_numbers:
        print(number)
```

Since they're iterable, we can also unpack them using `*`:

```
def cube(number):
        return number ** 3

numbers = [1, 2, 3, 4, 5]
cubed_numbers = map(cube, numbers)

print(*cubed_numbers, sep=", ")
```

We can also convert them to a normal collection if we like:

```
def cube(number):
        return number ** 3

numbers = [1, 2, 3, 4, 5]
cubed_numbers = list(map(cube, numbers))
```

`map` with multiple iterables

One of the really nice things about `map` is that is can handle several iterables at once.

When we provide more than one iterable, `map` takes a value from each one when it calls the provided function. This means the function gets called with multiple arguments. The order of those arguments matches the order in which we passed the iterables to `map`.

Let's say we have two lists of numbers: `odds` and `evens`. I want to add the first value in `odds` to the first values in `evens`, then I want

to add the second value of each collection, and so on. This is very easy to do with `map`.

```
def add(a, b):
        return a + b

odds = [1, 3, 5, 7, 9]
evens = [2, 4, 6, 8, 10]

totals = map(add, odds, evens)
print(*totals, sep=", ")  # 3, 7, 11, 15, 19
```

Note that the `add` function needs to be able to accept multiple values here, because `map` is always going to call it with two arguments: one values from `odds` and one from `evens`.

If the iterables of are differing lengths, `map` will stop as soon as it runs out of values, much like when using `zip`.

`map` with lambda expressions

`map` is frequently used for simple operations, which means it's often not worth defining a full blown function. Lambda expressions are often used instead, because they allow us to define a function inline while calling `map`.

For example, we can recreate the `cube` example above using a lambda expression like so:

```
numbers = [1, 2, 3, 4, 5]
cubed_numbers = map(lambda number: number ** 3, number
s)
```

If you need a reminder on how lambda expressions work, we covered them in day 16.

The `operator` module

While lambda expressions are great, we often end up using lambda expressions to duplicate the functionality of some operator. For example, `lambda number: number ** 3` is really just a way of using the `**` operator on each value.

Since this kind of lambda expression is so common, there's a module in the standard library called `operator` which contains function versions of all of the operators. It also includes some functions for making it easy to call methods or access values in collections.

Let's look at our `add` example again.

```
def add(a, b):
        return a + b

odds = [1, 3, 5, 7, 9]
evens = [2, 4, 6, 8, 10]

totals = map(add, odds, evens)
print(*totals, sep=", ")  # 3, 7, 11, 15, 19
```

We could have used a lambda expression here like this:

```
odds = [1, 3, 5, 7, 9]
evens = [2, 4, 6, 8, 10]

totals = map(lambda a, b: a + b, odds, evens)
print(*totals, sep=", ")  # 3, 7, 11, 15, 19
```

This is a bit messy though, in my opinion. It's not as clear as just writing `add`, but I also don't really want to have to define `add` if I don't have to.

`operator` has an `add` function ready to go, so we don't have to define it ourselves.

```
from operator import add
```

```
odds = [1, 3, 5, 7, 9]
evens = [2, 4, 6, 8, 10]

totals = map(add, odds, evens)
print(*totals, sep=", ")  # 3, 7, 11, 15, 19
```

Another really useful function in `operator` is `methodcaller`. `methodcaller` allows us to easily define a function that calls a method for us. We just have to provide the method name as a string.

For example, we can call the `title` method on a every name in a list like this:

```
from operator import methodcaller

names = ["tom", "richard", "harold"]
title_names = map(methodcaller("title"), names)
```

In my opinion, this is quite a lot nicer than a lambda expression approach:

```
names = ["tom", "richard", "harold"]
title_names = map(lambda name: name.title(), names)
```

There's a lot of neat things in the `operator` module, so I'd recommend you take a look at the documentation.

Conditional comprehensions

As I said at the start of this post, we can use comprehension for more than just performing an action for each item in an iterable, we can also perform filtering with comprehensions.

We do this by providing a condition at the end of our comprehension, and this condition determines whether or not an item makes it into our new collection. In cases where the condition evaluates to `True`, the item is added; otherwise, it gets discarded.

For example, let's say we have a set of numbers and I only want the even values. We can use a conditional comprehension to accomplish this:

```
numbers = [1, 56, 3, 5, 24, 19, 88, 37]
even_numbers = [number for number in numbers if number
% 2 == 0]
```

Here my filtering condition is `if number % 2 == 0`. So we're going to add any numbers which divide by `2` and leave a remainder of `0`.

This comprehension is equivalent to writing a `for` loop like this:

```
numbers = [1, 56, 3, 5, 24, 19, 88, 37]
even_numbers = []

for number in numbers:
        if number % 2 == 0:
                even_numbers.append(number)
```

We can do this filtering operation with any kind of comprehension, so we could do the same thing for a set comprehension, for example.

```
numbers = [1, 56, 3, 5, 24, 19, 88, 37]
even_numbers = {number for number in numbers if number
% 2 == 0}
```

We can also still perform modifications to the values as part of creating the new collection. The filtering condition is an augmentation of the usual comprehension syntax, and it doesn't limit anything we can normally do.

The `filter` function

Much like `map` is a functional analogue for "normal" comprehensions, `filter` performs the same role as a conditional comprehension.

Much like `map`, `filter` calls a function (known as a predicate) for every item in an iterable, and it discards any values for which that function returns a falsy value.

A **predicate** is a function that accepts some value as an argument and returns either `True` or `False`.

For example, we can rewrite our conditional comprehension from the example above like this:

```
numbers = [1, 56, 3, 5, 24, 19, 88, 37]
even_numbers = filter(lambda number: number % 2 == 0,
 numbers)
```

In this case we don't have an easy solution available in the `operator` module—though there is a `mod` function—so we have to either use a lambda expression, or we have to define a function to call.

In this case, I think it's worth defining a little helper function, because it makes things a lot more readable.

```
def is_even(number):
        return number % 2 == 0

numbers = [1, 56, 3, 5, 24, 19, 88, 37]
even_numbers = filter(is_even, numbers)
```

Just like `map`, `filter` gives us a lazy `filter` object, so the values are not calculated until we need them.

However, unlike `map`, the `filter` function can only handle a single iterable at a time. This isn't a big problem though, as we have many different ways of combining collections together, and we can pass in this combined collection to `filter`.

Using `None` with `filter`

Instead of passing in a function to `filter`, it's possible to use the value `None`. This tells `filter` that we want to use the truth values of the values directly, instead of performing some kind of comparison, or calculating something.

In this case, `filter` will keep all truthy values from the original iterable, and all falsy values will be discarded.

```
values = [0, "Hello", [], {}, 435, -4.2, ""]
truthy_values = filter(None, values)

print(*truthy_values, sep=", ")  # Hello, 435, -4.2
```

Exercises

1) Use `map` to call the `strip` method on each string in the following list:

```
humpty_dumpty = [
        "  Humpty Dumpty sat on a wall,  ",
        "Humpty Dumpty had a great fall;     ",
        "  All the king's horses and all the king's me
n ",
        "    Couldn't put Humpty together again."
]
```

Print the lines of the nursery rhyme on different lines in the console.

Remember that you can use the `operator` module and the `methodcaller` function instead of a lambda expression if you want to.

2) Below you'll find a tuple containing several names:

```
names = ("bob", "Christopher", "Rachel", "MICHAEL", "j
essika", "francine")
```

Use a list comprehension with a filtering condition so that only names with fewer than 8 characters end up in the new list. Make sure that every name in the new list is in title case.

3) Use `filter` to remove all negative numbers from the following range: `range(-5, 11)`. Print the remaining numbers to the console.

You can find our solutions to the exercises here.

Additional Resources

There's a version of `filter` in the `itertools` module called `filterfalse`. It works just like `filter`, but only items for which the predicate returns a falsy value are kept.

In essence, it gives us the set of values that `filter` would throw away.

You can find documentation for `filterfalse` here.
