teclado

ADVANCED PYTHON

Day 15: Comprehensions

Welcome to day 15 of the 30 Days of Python series! Today we're going to be looking at a new type of syntax called comprehensions.

Comprehensions allow us to create a collection from some other iterable in a very succinct way. Very often they'll save us lines and lines of code, and they help to cut down on needless boilerplate.

List comprehensions

There are several different types of comprehension, but the most commonly used are list comprehensions.

List comprehensions are used to create a new list from some other

iterable. It might be another list, or maybe even something like a `zip` object.

Let's look at an example of when a list comprehension might be useful. Imagine I have a list of names like this:

```
names = ["mary", "Richard", "Noah", "KATE"]
```

Right now they're not very consistent. The names are written in different cases, which can easily happen if we're grabbing data from users.

I want to make sure the names are all in title case, so I'm going to iterate over the list and create a new title case string from each name.

```
names = ["mary", "Richard", "Noah", "KATE"]
processed_names = []

for name in names:
        processed_names.append(name.title())
```

Once we're done, `processed_names` is going to contain all of our title case strings, just like we wanted.

There are a couple of things I don't like about this approach though.

1. We have to use a new name for this second list. We can't simply throw away the old `names` list and replace it, because we're having to iterate over that list to process the values.
2. A lot of this code feels a bit unnecessary. We're really just trying to create a new list with different values, so having to create an empty list and append values feels like the language is getting in the way of doing something simple.

This is where the comprehension syntax comes in. Before we talk about the syntax in detail, let's look at the example above using a list

the syntax in detail, let's look at the example above using a list comprehension instead of a traditional loop:

```
names = ["mary", "Richard", "Noah", "KATE"]
processed_names = [name.title() for name in names]
```

Better yet, we can reuse `names` instead of assigning to `processed_names`:

```
names = ["mary", "Richard", "Noah", "KATE"]
names = [name.title() for name in names]
```

Pretty cool!

For comprehensions like this, there are two main parts of the structure: the value we want to add to the new list, and a loop definition.

In the example above, the value we want to add is `name.title()`, and the loop we've defined is `for name in names`.

We can think of it as saying, "Put `name.title()` in the new list for every `name` in `names`".

Just like with regular loops, we can define more complex comprehensions where we have multiple loop variables.

For example, we can take some code like this:

```
names = ("mary", "Richard", "Noah", "KATE")
ages = (36, 21, 40, 28)

people = []

for name, age in zip(names, ages):
        person_data = (name.title(), age)
        people.append(person_data)
```

And we can rewrite this using a comprehension:

```
names = ("mary", "Richard", "Noah", "KATE")
ages = (36, 21, 40, 28)

people = [(name.title(), age) for name, age in zip(names, age
s)]
```

In both cases, the `people` lists ends up containing the following:

```
[('Mary', 36), ('Richard', 21), ('Noah', 40), ('Kate', 28)]
```

Note that in this example I've used tuples for the `names` and `ages` collections, because it's important this data stays in a specific order. We wouldn't want somebody to sort these collections or anything like that, because we could end up matching a name to the wrong age.

Style note

Sometimes comprehensions can get fairly long, especially if we have long variable names. In those cases, it can be helpful to break the comprehension up over several lines, like this:

```
people = [
        (name.title(), age)
        for name, age in zip(names, ages)
]
```

A logical way to break this up is to have the value we're adding to the list on one line, and the loop definition on the next.

It's also well worth noting that comprehensions are not always the right tool for the job, and they're not a wholesale replacement for regular for

loops. If we have some complex logic to perform, regular loops can be much clearer, because we can break those operations up into smaller steps, and use variable names to signpost what is happening at various points.

None of this is possible in a comprehension, so they're generally only suitable for simpler operations.

Set comprehensions

A lesser known type of comprehension is the set comprehension. This works just like a list comprehension, but it produces a set, rather than a list.

The internal structure of the comprehension is identical to the list comprehension, we just need to surround everything in curly braces, rather than square brackets.

```
names = ["mary", "Richard", "Noah", "KATE"]
names = {name.title() for name in names}
```

Now `names` is a set containing four title case strings:

```
{'Mary', 'Richard', 'Noah', 'Kate'}
```

Dictionary comprehensions

The final type of comprehension we have available to us is the dictionary comprehension. Like set comprehensions and regular dictionaries, dictionary comprehensions are surrounded by curly braces.

Dictionary comprehensions are a little different from the list and set comprehensions that we've seen already though, because dictionaries need both a key and a value for each item.

The syntax we use for the key value pair is exactly the same as when we define a dictionary normally. First we have the key, followed by a colon (`:`), and then the value. This whole structure is what goes in the value position of our comprehension.

For example, let's say we have a loop like this:

```
student_ids = (112343, 134555, 113826, 124888)
names = ("mary", "Richard", "Noah", "KATE")

students = {}

for student_id, name in zip(student_ids, names):
        student = {student_id: name.title()}
        students.update(student)
```

Here we have two tuples containing related information about some students. In the first tuple we have some student id numbers, and in the second tuple we have the names of the students in same order.

We zip these collections together, so that we have something like this in our `zip` object:

```
(112343, "mary"), (134555, "Richard"), (113826, "Noah"), (124
888, "KATE")
```

We then use these pairs to create new dictionaries so that we can update the `students` dictionary with new key value pairs. Remember that `update` can accept a dictionary as an argument, and it will merge the keys and values of that dictionary into another dictionary.

The result is a dictionary like this:

```
{112343: 'Mary', 134555: 'Richard', 113826: 'Noah', 124888:
'Kate'}
```

We can produce the same dictionary with a dictionary comprehension like so:

```
student_ids = (112343, 134555, 113826, 124888)
names = ("mary", "Richard", "Noah", "KATE")

students = {
        student_id: name.title()
        for student_id, name in zip(student_ids, names)
}
```

Once again, we have the value we want to add to the new collection, which for a dictionary comprehension is a key value pair.

```
student_id: name.title()
```

This is followed by a loop definition which determines where the original values are coming from:

```
for student_id, name in zip(student_ids, names)
```

All of this is then placed within curly braces to indicate that we're using a dictionary comprehension:

```
students = {
        student_id: name.title()
        for student_id, name in zip(student_ids, names)
}
```

Comprehensions and scope

One thing you may be wondering is, how come we can use a variable name in our comprehension and assign to the same variable name? Aren't we replacing values while we're accessing them? And isn't that dangerous?

You may have also noticed we can't refer to the variables we define in the comprehension's loop from outside of the comprehension.

For example, with a regular loop, we can do something like this:

```
names = ["Mary", "Richard", "Noah", "Kate"]
names_lower = []

for name in names:
        names_lower.append(name.lower())

print(name)  # This refers to the name variable we defined in
the loop
```

`name` is just a normal variable, and once the loop is done, that variable still exists. Its value is going to be the last value it was assigned in the loop. In the case above, `name` would refer to the string, `"Kate"` .

If we use a comprehension of the other hand, we can't do this:

```
names = ["Mary", "Richard", "Noah", "Kate"]
names_lower = [name.lower() for name in names]

print(name)  # NameError
```

Referring to `name` gives us a `NameError` , because `name` isn't defined:

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
```

```
    print(name)  # NameError
NameError: name 'name' is not defined
```

So, what exactly is going on here?

The reason for these differences is that comprehensions are actually functions. When we write something like this:

```
names = ["Mary", "Richard", "Noah", "Kate"]
names_lower = [name.lower() for name in names]
```

Behind the scenes, Python is running a function similar to this:

```
def temp():
        new_list = []

        for name in names:
                new_list.append(name.lower())

        return new_list
```

It's just a regular `for` loop, like we were using before, only this `for` loop is running inside a function.

If you remember back to day 13, we talked about the fact that when we call a function, a new namespace is created to store the variable names we define inside the function. This dictionary of names and values is only accessible within the function.

We can now see why the variables inside comprehensions cannot be accessed outside of the comprehension. They only exist while the behind-the-scenes function is running, and they're only available inside that function.

Similarly, we can see why we can assign to a variable name while seemingly using it within the comprehension.

Inside the function, we're actually creating an entirely new list, and we're assigning this list to a temporary name. We're therefore not modifying the original iterable while we populate our new list. Only when we're done do we return the new list and potentially replace the existing value assigned to the variable.

We generally don't have to worry about this function that Python makes and runs for us, but it's nice to know that comprehensions are really just regular `for` loops at their heart.

The fact that the code inside them is encapsulated in this function is also very useful, because it means our comprehensions don't affect the code around them in the way that `for` loops can. After all, if we use a name as a loop variable, we could be overwriting an existing value associated with that name. That's not an issue in comprehensions.

Exercises

1) Convert the following for loop into a comprehension:

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

for number in numbers:
        squares.append(number ** 2)
```

2) Use a dictionary comprehension to create a new dictionary from the dictionary below, where each of the values is title case.

```
movie = {
        "title": "thor: ragnarok",
```

```
        "director": "taika waititi",
        "producer": "kevin feige",
        "production_company": "marvel studios"
}
```

Remember that iterating over a dictionary only gives us the keys by default. You can use the `items` method to get the keys and the values. See day 10 for more details.

You can find our solutions to the exercises here.

Additional Resources

We can also use comprehensions for filtering by adding conditional logic to the comprehension. We have a post discussing this in more detail on our blog.

We'll also be covering this in day 20.
