WORKING WITH DATA

Day 9: Unpacking, Enumeration, and the zip Function

Welcome to day 9 of the 30 Days of Python series! Today we're going to look at a really useful iteration technique called unpacking. We're also going to be exploring a couple of new functions: `enumerate` and `zip`.

If you missed yesterday's post, we covered a very important structure called a `while` loop, so I'd recommend taking a look if that's something you're not familiar with.

Unpacking

Unpacking is generally used to perform several assignments at once,

by extracting the individual values from some iterable. This process is also called destructuring.
Let's look at a very simple case. Maybe I have a movie record, which is a tuple, and it contains the movie title, the director's name, and the year of release. What I would like to do, is assign this tuple's values to three variables: `title`, `director`, and `year`.

With our current knowledge, we have to do something like this:

```
movie = ("12 Angry Men", "Sidney Lumet", 1957)

title = movie[0]
director = movie[1]
year = movie[2]
```

This is a bit tedious though, and thankfully there's a much better way. Instead of writing separate assignments, and accessing indices, we can simply write this:

```
movie = ("12 Angry Men", "Sidney Lumet", 1957)

title, director, year = movie
```

What's happening here is Python sees that we have three names that we want to assign to, and the tuple we're assigning to it has three elements. It therefore grabs an item from the tuples, one at a time, and it assigns these values to the names in order.

The number of names must match up with the number of elements in the iterable, otherwise we're going to get an exception like this:

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    title, director, year = movie
ValueError: not enough values to unpack (expected 3, g
ot 2)
```

We're not limited to just tuples here: we can use lists, or any other

iterable type. This is extremely useful, and we can use this technique in loops to great effect, as we're about to see!

Unpacking in for loops

Let's go back to our movie library example, where we have a list containing several movie tuples:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]
```

When we were first looking at for loops, we had an example where we printed the values from each of these tuples in a nice way for the user, so that they could see all of their movies.

The loop looked like this:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
```

```
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]

for movie in movies:
        print(f"{movie[0]} ({movie[2]}), by {movie[1]}
")
```

This loop works totally fine, but one thing which isn't great is that the values are really opaque. What is `movie[0]`? We really have to go back to the movies list and figure this out. It would be much better if we could use nice, descriptive names here instead.

Let's think about what's going on when we iterate over `movies`. For each iteration, `movies` is going to give us a single tuple, and we're going to assign this value to `movie`.

If we think about it, that's very similar to doing this:

```
movie = ("Memento", "Christopher Nolan", 2000)
```

We have a name, `movie`, and we're assigning a tuple to it. In this case, we have the second tuple from the `movies` list.

However, we just saw an example of a tuple like this being unpacked into three variables, like so:

```
title, director, year = ("Memento", "Christopher Nola
n", 2000)
```

As it turns out, we can do exactly the same thing in our for loop:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]

for title, director, year in movies:
        print(f"{title} ({year}), by {director}")
```

For each iteration of the loop, `movies` gives us a single tuple. Python sees that the tuple has three values in it, and we've defined three names. It therefore takes the first item in the tuple and assigns it to the name `title`; it takes the second item and assigns it to `director`; and it takes the third value and assigns it to `year`.

By using unpacking here, we've made our `print` call so much more readable, because instead of referring to indices, we have clear, descriptive names.

If you want to learn more about this technique, we have a blog post which goes into more detail.

Enumeration

One common application for unpacking is when using a function called `enumerate`

called `enumerate`.

Before we look at the function itself, I want to add a piece of functionality to our printing loop. As well as the movie information, I want to start each line with a number, which represents the order of the movies.

In other words, I want output like this:

```
1. Eternal Sunshine of the Spotless Mind (2004), by Mi
chel Gondry
2. Memento (2000), by Christopher Nolan
3. Requiem for a Dream (2000), by Darren Aronofsky
```

How would we go about this?

Through some means, we're going to need to generate some kind of counter. Here is a very common solution to this problem that I see all the time:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]

for index in range(len(movies)):
        print(f"{index + 1}. {movies[index][0]} ({movi
```

```
        print(f"{index + 1}. {movies[index][0]} ({movi
es[index][2]}), by {movies[index][1]}")
```

To be very clear, while this kind of solution is common, it is not a good solution. This clearly very hard to read, and we've lost all of our great names.

Just for completeness, let's talk about how this works, because I guarantee you will see solutions like this at some point.

First, we're calling this function called `len` to find how many movies are in the `movies` list. In our case, there are `3` movies, so `len(movies)` is going to evaluate to `3`.

This value, `3`, is then being passed into `range`. As we saw in day 6, if we pass a single value to `range` it gives us a series of values from `0` up to (but not including) the number we passed in. Our `range` therefore looks like this:

```
0, 1, 2
```

`range` is an iterable, and the first value it's going to give us is `0`, which we're assigning to this name `index`. We then use this value to look up a movie tuple in `movies`.

Remember that `movies[0]` is going to give us the first movie in the list, and `movies[1]` gives us the second value, etc.

As we go through the `range` values, the number increases, and we end up looking at each movie in turn. For each movie tuple, we're accessing the data we want using another subscription expression (the square bracket syntax), just like we did before.

The `index` is also serving as our counter. For the first iteration, it's `0`, so we just need to add `1` to it to get the number `1` printed next to our first movies. It then increases by `1` each time, giving us an incrementing counter at the start of our print output.

Okay, so this isn't a great solution, but what's a better way? Well, we

could define a counter outside of the loop, and then we could increment the counter for each iteration:

```
movies = [ ... ]
counter = 1

for title, director, year in movies:
        print(f"{counter}. {title} ({year}), by {director}")
        counter += 1
```

We haven't seen this `+=` syntax before, but `counter += 1` is a shorthand for writing this:

```
counter = counter + 1
```

This is a much better solution, but we have this book keeping variable now, and I don't really like that. A better solution, in my opinion, is using `enumerate`.

`enumerate` is a built in function that generates a counter alongside the values in an iterable. What we end up with is a series of tuples, where the first item in each tuple is the counter, and the second value is the original item from the iterable.

Let's look at a simple example first. I just want to create a counter alongside a list of names. We can do this like so:

```
names = ["Harry", "Rachel", "Brian"]

for counter, name in enumerate(names):
        print(f"{counter}. {name}")
```

What `enumerate` contains in this case is a series of tuples like this:

```
(0, "Harry"), (1, "Rachel"), (2, "Brian")
```

And our output looks like this:

```
0. Harry
1. Rachel
2. Brian
```

This isn't quite what I want, because I want the values to start from `1`. This is an easy fix, because we can set a starting value for the counter when we call `enumerate`:

```
names = ["Harry", "Rachel", "Brian"]

for counter, name in enumerate(names, start=1):
        print(f"{counter}. {name}")
```

Now we get the output we wanted:

```
1. Harry
2. Rachel
3. Brian
```

Let's go back to our original example with the movies. Things are not quite as easy here, because each of our items is itself a tuple. We could just do this:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
```

```
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]

for counter, movie in enumerate(movies, start=1):
        print(f"{counter}. {movie[0]} ({movie[2]}), by
{movie[1]}")
```

Here the result of `enumerate` is something like this:

```
(1, ("Eternal Sunshine of the Spotless Mind", "Michel
 Gondry", 2004)),
(2, ("Memento", "Christopher Nolan", 2000)),
(3, ("Requiem for a Dream", "Darren Aronofsky", 2000))
```

So `counter` takes the value of each number in the first position (`1` , `2` , and `3`). And `movie` takes the value of each tuple in the second position (such as `("Memento", "Christopher Nolan", 2000)`).

However, I still want to be able to unpack this movie tuple into these descriptive variable names we had before. But how do we unpack something like this:

```
(2, ("Memento", "Christopher Nolan", 2000))
```

All we have to do is match the structure of the thing we want to unpack. Here we have a two element tuple, where the second item is a three element tuple. So we can unpack this structure above like so:

```
counter, (title, director, year) = 2, ("Memento", "Chr
istopher Nolan", 2000)
```

In our loop, we just need to do the same thing:

```
movies = [ ... ]

for counter, (title, director, year) in enumerate(movi
es, start=1):
        print(f"{counter}. {title} ({year}), by {direc
tor}")
```

Note that the brackets here **are** really important. If we write this,

```
movies = [ ... ]

for counter, title, director, year in enumerate(movie
s, start=1):
        print(f"{counter}. {title} ({year}), by {direc
tor}")
```

we're going to get a `ValueError`, because we've provided four names, but only two values: the counter, and the movie tuple. By using these parentheses, we're specifying that we want this second value (the tuple) broken down into three separate names.

The `zip` function

`zip` is an extremely powerful and versatile function used to combined two or more iterables into a single iterable.

For example, let's say that we have two lists like this:

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]
```

`zip` will allow us to turn this into a new iterable which contains the following:

```
("Paul", "Fluffy"), ("Andrea", "Bubbles"), ("Marta",
```

```
"Captain Catsworth")
```

In essence, it takes the first item from each iterable, and puts together them in a tuple. Then it takes the second item from each iterable, and so on, until one of the iterables runs out of values. We'll come back to this point later, because it's really important.

To use `zip`, all we have to do is call the function and pass in the iterables we want to zip together.

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]

pets_and_owners = zip(pet_owners, pets)
```

If we want to zip three or even more iterables together, we can just keep passing more and more items to `zip` when we call it.

Much like `range`, `zip` is lazy, which means it only calculates the next value when we request it. We therefore can't print it directly, but we can convert it to something like a list if we want to see the output:

```
print(list(pets_and_owners))

# [('Paul', 'Fluffy'), ('Andrea', 'Bubbles'), ('Mart
a', 'Captain Catsworth')]
```

Using `zip` in loops

Another very common way to use `zip` is to iterate over two or more iterables at once in a for loop.

Let's go back to our pet owners example, but now I want to print some output which describes who owns which pet.

We can use `zip` and a bit of destructuring to do this in a really clear way, because we get to use nice clear variable names in the loop:

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]

for owner, pet in zip(pet_owners, pets):
        print(f"{owner} owns {pet}.")
```

The common alternative to using `zip` is the nasty `range + len` pattern we saw earlier with `enumerate`. I'd recommend avoiding that at all costs!

A caveat for when using `enumerate` and `zip`

One thing you should be aware of when it comes to `enumerate` and `zip` is that they are consumed when we iterate over them. This generally isn't a problem when we use them directly in loops, but it can sometimes trip up newer developers when they assign a `zip` or `enumerate` object to a variable.

Here is an example where we assign the result of calling `zip` to a variable:

```
movie_titles = [
        "Forrest Gump",
        "Howl's Moving Castle",
        "No Country for Old Men"
]

movie_directors = [
        "Robert Zemeckis",
        "Hayao Miyazaki",
        "Joel and Ethan Coen"
]

movies = zip(movie_titles, movie_directors)
```

We can iterate over movies without any problems:

We can iterate over the movies without any problems:

```
for title, director in movies:
        print(f"{title} by {director}.")
```

However, if we now try to use `movies` again, we'll find that it's empty. Try running the code below to see this:

```
movie_titles = [
        "Forrest Gump",
        "Howl's Moving Castle",
        "No Country for Old Men"
]

movie_directors = [
        "Robert Zemeckis",
        "Hayao Miyazaki",
        "Joel and Ethan Coen"
]

movies = zip(movie_titles, movie_directors)

for title, director in movies:
        print(f"{title} by {director}.")

movies_list = list(movies)

print(f"There are {len(movies_list)} movies in the col
lection.")
print(f"These are our movies: {movies_list}.")
```

If you try to iterate over `movies` after the initial loop, you'll also find that it contains no values.

The reason that this happens is because `zip` and `enumerate` produce something called an *iterator*. We're not going to be talking about iterators in any depth in this series, as iterators are an advanced topic, but one key feature of iterators is that they're consumed when we request their values. This is actually a really useful feature, but it's also a common source of bugs if you're not

familiar with this behaviour.

An easy way to bypass this limitation is to just convert the iterator to a non-iterator collection, like a list or tuple.

```
movie_titles = [
        "Forrest Gump",
        "Howl's Moving Castle",
        "No Country for Old Men"
]

movie_directors = [
        "Robert Zemeckis",
        "Hayao Miyazaki",
        "Joel and Ethan Coen"
]

movies = list(zip(movie_titles, movie_directors))
```

Now we can access the values in `movies` as many times as we like.

Exercises

1) Below is some simple data about characters from BoJack Horseman:

```
main_characters = [
    ("BoJack Horseman", "Will Arnett", "Horse"),
    ("Princess Carolyn", "Amy Sedaris", "Cat"),
    ("Diane Nguyen", "Alison Brie", "Human"),
    ("Mr. Peanutbutter", "Paul F. Tompkins", "Dog"),
    ("Todd Chavez", "Aaron Paul", "Human")
]
```

The data contains the character name, the voice actor or actress who plays them, and the species of the character.

Write a for loop that uses destructuring so that you can print each tuple in the following format:

```
BoJack Horseman is a horse voiced by Will Arnet.
```

Note that you're going to have to change the species information to lowercase when you print it. If you need a reminder on how to do this, we covered it in day 3 of the first week.

2) Unpack the following tuple into 4 variables:

```
("John Smith", 11743, ("Computer Science", "Mathematic
s"))
```

The data represents a student's name, their student id number, and their major and minor disciplines in that order.

3) Investigate what happens when you try to zip two iterables of different lengths. For example, try to zip a list containing three items, and a tuples containing four items.

You can find our solutions to the exercises here.

Project

Once you're done with the exercises, we have another project for you today. This time we're writing a short program to validate credit cards!

Additional resources

If you want to learn more about unpacking, we have a blog post which goes into more detail.

We also have a blog post on `enumerate`, which you can find here, and one on `zip`, which you can read here.

In addition to all of these, we have a post talking about another function called `zip longest`. This allows us to zip differently sized

collections without losing data, but it requires some techniques we haven't looked into yet, such as importing code from other files. If you're interested, you can find that post here.
