A CHALLENGE: FIZZBUZZ

Day 4: Basic Python Collections

Welcome to day 4 of the 30 Days of Python series! In this post we're going to be introducing some new collections types: the list and the tuple.

If you missed day 3, I'd recommend checking it out if you need a refresher on any of the following:

- String concatenation
- String interpolation using `format` and f-strings
- Processing strings using `lower` , `upper` , `capitalize` , `title` , and `strip`
- Documenting your code using comments

Also we've created a short recap of Days 1, 2, and 3 that you can view here to refresh your memory.

Lists

The first new collection we're going to look at are lists.

Simply put, lists are containers for other values. Unlike strings, which are collections of just characters, lists can store whatever values we like. We can even mix different types of values if we want.

We define a list using square brackets like this:

```
names = ["John", "Alice", "Sarah", "George"]
```

Each value is placed inside the square brackets, separated by commas.

In this case, we've defined a list all on one line. Here it's fine, but sometimes this can get a little hard to read. For example, what if we're trying to store a list of movie titles, and we have strings like, `"Eternal Sunshine of the Spotless Mind"` ? We're very quickly going to end up with code exceeding the width of our screens. The same goes for if we have lots and lots of values.

When we run into a situation like this, we can split the list across multiple lines like so:

```
movie_titles = [
    "Eternal Sunshine of the Spotless Mind",
    "Memento",
    "Requiem for a Dream"
]
```

As I mentioned previously, we can mix whatever types of value we want in a list, you don't have to have just strings, or just integers.

```
friend_details = ["John", 27, "Web Developer"]
```

It’s also possible to define a list with no content, which is represented by an empty pair of square brackets `[]`.

We can print an entire list by passing the list to the `print` function, like so:

```
friend_details = ["John", 27, "Web Developer"]

print(friend_details)
```

However, usually we’re interested in the values inside a list, rather than the list as a whole, so let’s look at how to access list values.

Style note

Resist the temptation to do something like this:

```
movie_titles = [
    "Eternal Sunshine of the Spotless Mind",
    "Memento", "Requiem for a Dream"
]
```

It may be shorter than putting a different value on each line, but it makes it much harder to see how many values are in the list, and it can also make it harder to see the values themselves. For example, there’s a fairly subtle difference between:

```
"Memento", "Requiem for a Dream"
```

and

```
"Memento, Requiem for a Dream"
```

And this is very easy to miss in longer lists.

When we're writing lists all on one line, it's also good practice to put a space after the commas. One again, this is *really* important for readability.

Compare this:

```python
numbers = [343.4,545.98,4,954.03,3.5]
```

To this:

```python
numbers = [343.4, 545.98, 4, 954.03, 3.5]
```

Accessing values in a list

Each value in a list is indexed according to its position in the list. The item in the first position is at index `0`; the item at the second position is at index `1`; and so on.

We can access values in a list using these indices, and we generally do this with a piece of syntax called a *subscription expression*.

Subscriptions expressions are used for accessing values in many collections. For sequences, they are used for accessing elements by index.

It looks something like this:

```python
names = ["John", "Alice", "Sarah", "George"]

print(names[2])  # Sarah
```

In order to retrieve the value `"Sarah"` from `names`, we write the name of the list we want to access, and then inside some square brackets we put the index of the item we want to retrieve.

It's also possible to refer to a *negative* index, which allows us to work from the end of the list. In that case, the item at index `-1` is the last item; the item as index `-2` is the penultimate item; and so on.

```
names = ["John", "Alice", "Sarah", "George"]

print(names[-1])  # George
```

This can be really useful for when we want the final item in a list, as it saves us having to do any calculations to find out the index of the final item.

Adding items to a list

One neat feature of lists is that they're mutable: we can change the values inside. This means we can add values, remove them, replace them, reorder them, etc.

First, let's focus on adding items to a list using the `append` method. As the name might imply, `append` lets us add an item to the end of a list. Once again, we need to use the dot notation to call `append`, and we put the value we want to add inside the parentheses when we call it.

```
names = ["John", "Alice", "Sarah", "George"]
names.append("Simon")

print(names[-1])  # Simon
```

We can also add another item (or items) to a list using the `+` operator. In this case, both operands **must** be lists.

The approach is also somewhat different, as we have to perform an assignment as part of the operation.

```
names = ["John", "Alice", "Sarah", "George"]
names = ["John", "Alice", "Sarah", "George"]
names = names + ["Simon"]

print(names[-1])  # Simon
```

This can sometimes be handy, because it doesn't modify the original list, as the operation yields a new list instead. `append`, on the other hand, modifies the existing list.

In addition to `append` method and the `+` operator, we also have a method called `extend`, which allows us to add multiple items to the end of an existing list.

These are all great for adding an item to the end of a list, but what if we want to add an item in the middle? For that, we have the `insert` method.

`insert` is a little more complicated than the options we've looked at so far, as it needs two pieces of information. First we need to tell it where we want to insert the value, and second we need to tell it what we want to insert.

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

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

At the moment, we're missing the value `3` in this sequence, and we want to insert this value at index `2`. We therefore need to call `insert` like so:

```
numbers = [1, 2, 4, 5]
numbers.insert(2, 3)

print(numbers)  # [1, 2, 3, 4, 5]
```

As you can see, the other values were shuffled right to make room for our new addition.

One thing you may be wondering is, what happens if we specify an index outside of the defined list? Python just puts the item as the end instead:

```
numbers = [1, 2, 4, 5]
numbers.insert(7, 3)

print(numbers)  # [1, 2, 4, 5, 3]
```

Removing items from a list

Just like with adding items, there are several options available to use when it comes to removing items from a list.

First up, we have the `remove` method. `remove` once again uses the dot syntax we've seen a number of times now, and it removes the first item that matches the value we pass in.

For example, if we have a list like this:

```
names = ["John", "Sarah", "Alice", "John"]
```

We can remove the first `"John"` by calling `remove` :

```
names = ["John", "Sarah", "Alice", "John"]
names.remove("John")

print(names)  # ['Sarah', 'Alice', 'John']
```

The second instance of `"John"` is left in tact, and everything shuffles left so we don't have a hole in our list.

Sometimes we don't necessarily know the value we want to remove, but we know *where* the value is in our list. In cases like this, we have a couple of options.

First, we can use the `del` keyword. `del` can be used to delete anything we want, including a whole list, but if we use a subscription expression, we can use it to remove an item at a specific index.

```python
names = ["John", "Sarah", "Alice", "Mike"]
del names[0]

print(names)  # ['Sarah', 'Alice', 'Mike']
```

The main alternative we have to using `del` is `pop` . By default `pop` is going to remove the last item in the list, but we can optionally pass in an index as an argument to remove a different item instead.

```python
names = ["John", "Sarah", "Alice", "Mike"]
names.pop()

print(names)  # ['John', 'Sarah', 'Alice']

names.pop(1)

print(names)  # ['John', 'Alice']
```

One of the really useful things about `pop` is that the method call expression evaluates to the item we removed from the list. We can therefore use this to remove an item which we plan to use for some future operation:

```python
names = ["John", "Sarah", "Alice", "Mike"]
last_in_line = names.pop()

print(names)         # ['John', 'Sarah', 'Alice']
print(last_in_line)  # Mike
```

Finally, we have `clear` . This one is pretty straightforward, it's just going to remove everything inside a given list:

```
names = ["John", "Sarah", "Alice", "John"]
names.clear()

print(names)  # []
```

Tuples

So far we've spoken a lot about lists, but what about tuples?

Tuples are very similar to lists in that they're containers for other values, but there are some important differences that we'll discuss later one.

All we need to define a tuple is a series of comma separated values:

```
names = "John", "Sarah", "Alice"
```

More often, however, we'll see tuples written like this:

```
names = ("John", "Sarah", "Alice")
```

Much of the time, these parentheses are optional—it's the commas that are important—but there are cases where we really need them to distinguish the tuple contents from other comma separated values.

For example, it's totally legal (and common) to put tuples in a list. Maybe we want to store movie names and release dates together in a list of movies like this:

```
movies = [
        ("Eternal Sunshine of the Spotless Mind", 2004
),
        ("Memento", 2000),
        ("Requiem for a Dream", 2000)
]
```

Here we have a list containing three tuples, where each tuple contains two items.

If we didn’t use the parentheses around the tuples here, there’d be no way to distinguish the tuples from one another, as the list also uses commas to separate values.

Without the brackets, we'd have this:

```
movies = [
        "Eternal Sunshine of the Spotless Mind", 2004,
        "Memento", 2000,
        "Requiem for a Dream", 2000
]
```

Remember that Python doesn't care much for line breaks inside a list, so that is the same as this—a list with 6 elements:

```
movies = [
        "Eternal Sunshine of the Spotless Mind",
  2004,
        "Memento",
  2000,
        "Requiem for a Dream",
  2000
]
```

To avoid issues like this, the parentheses are mandatory.

One slightly awkward bit of syntax we have to remember is for defining tuples with just a single item. Remember that the commas are what make a tuple, so we can’t just write something like this:

```
name = ("John")
```

That’s just a string wrapped in parentheses. If we want to make this a tuple, we have to write this:

```
name = "John",
```

Or this:

```
name = ("John",)
```

Both are pretty ugly, but thankfully this isn't something we have to write all that often.

Accessing values in a tuple

Much like with lists, tuples are ordered collections, where each item can be accessed by index based on their position in the collection.

We even use the same subscription expression syntax to access items in a tuple.

At this point it's clear that tuples and lists share a lot of similarities, so maybe you're asking yourself, why do we have both?

Tuples vs lists

One of big differences between tuples and lists is that tuples are immutable. We can't change them once we define them.

This means you won't find any `pop` or `append` methods for tuples, and the `del` keyword isn't going to allow you to remove values using an index.

One thing that will work is the `+` operator, but if we remember back to our discussion of `+` with lists, this operator gives us a new collection. If we use it with a tuple, the original tuple is going to remain unchanged: we've just created a new one.

Note that you can only use `+` to join two tuples.

The fact that tuples can't change is a very desirable property, because it allows us to define a consistent structure for their contents, which means we use the values like cells in a table row. Let me show you what I mean.

Let's say we have three pieces of information we want to store for each movie in our collections. We want the title of the movie, the director's name, and the year of release.

I might define my `movies` collection 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 each tuple, I've used the same order for the data, which we can think of as representing a row in a table like this: TitleDirectorRelease YearEternal Sunshine of the Spotless MindMichel Gondry2004MementoChristopher Nolan2000Requiem for a DreamDarren Aronofsky2000 Because the tuples can't be changed, we can be sure that the data is always going to match this convention that we've defined. The item at the first index is always going to be the movie title. The item in the second index is always going to be the director name.

Compare this to a list. I can't be sure of the order anymore, because for all I know, the list could have had several items added, several removed, and the list could have been alphabetically sorted. It might even be empty. We just don't know, and that can sometimes be a problem.

One the other hand, being able to modify data is very, very useful. For example, if we'd created `movies` as a tuple, we can no longer add other movies to the collection. That would be no good if we were creating something like a movie library, where users may want to add new movies later on.

Neither one is better than the other: it's about choosing the right tool for the job.

Accessing values in nested collections

We've now had a few instances of nested collections—tuples inside a list, for example—so we need to briefly talk about how to get items out of these inner collections.

Remember that when we write something like `my_list[0]`, this is a subscription expression. The value it gives us is the value at the requested index in the collection we specified.

So if we have a collection like this:

```
movies = [
        ("Eternal Sunshine of the Spotless Mind", 2004
),
        ("Memento", 2000),
        ("Requiem for a Dream", 2000)
]
```

And we write:

```
movies[0]
```

```
movies[0]
```

The item we get back is the tuple,

```
("Eternal Sunshine of the Spotless Mind", 2004)
```

So if we want the title of this movie, which we know is at index `0`, we can add another set of square brackets like so:

```
movies[0][0]  # "Eternal Sunshine of the Spotless Mind"
```

This is equivalent to doing something like this:

```
movie = movies[0]  # ("Eternal Sunshine of the Spotless Mind", 2004)
movie[0]           # "Eternal Sunshine of the Spotless Mind"
```

We have two subscription expressions. The first yields the tuple at index `0` of `movies`, and then the second yields the value at index `0` of that tuple.

Exercises

1. Create a `movies` list containing a single tuple. The tuple should contain a movie title, the director's name, the release year of the movie, and the movie's budget.
2. Use the `input` function to gather information about another movie. You need a title, director's name, release year, and budget.
3. Create a new tuple from the values you gathered using `input`. Make sure they're in the same order as the tuple you wrote in the `movies` list.
4. Use an f-string to print the movie name and release year by accessing your new movie tuple.

5. Add the new movie tuple to the `movies` collection using `append`.
6. Print both movies in the `movies` collection.
7. Remove the first movie from `movies`. Use any method you like.

You can find solutions for these exercises here.

Additional Resources

We have a dedicated post on the `extend` method for lists which you can find on our blog.