teclado

WORKING WITH DATA

Day 7: split, join, and Slices

Welcome to day 7 of the 30 Days of Python series! Today we're returning to collections, and we're mainly going to talk about converting between strings and other collection types like lists and tuples. We're going to learn how to grab a subset of a collection using slicing.

If you missed day 6, I'd recommend you check it out if you don't know any of the following:

- `for` loops
- The `break` statement
- The `range` function

Converting tuples and lists to strings

Tuples and lists have a string representation, but it's often not what we want. If we try to pass a list to `str` to create a new string, like so:

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

We're going to see something like this if we try to print it:

```
[1, 2, 3, 4, 5]
```

Some of you may have noticed that this is exactly what we get if we try to print a list anyway. That's because when we pass a list to `print`, `print` converts the list to a string for us.

In terms of code, this is what you're ending up with. A string containing the square brackets, the numbers, and the commas:

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

In this case, however, what I actually want is something like this:

```
1, 2, 3, 4, 5
```

For example, maybe I have a list of names which describes who worked on some project, and I want to print these names out as part of a sentence:

```
The people who worked on this project are: Mike, Sofia, Hele
n.
```

What I don't want is this:

What I don't want, is this:

```
The people who worked on this project are: [Mike, Sofia, Hele
n].
```

The solution to our problem is to the `join` method.

We call `join` on a string, and that string is what we want to place between the items in the collection we want to glue together. In the example above, we just want to join the values using a comma and a space, so our code would look like this:

```
project_authors = ["Mike", "Sofia", "Helen"]
project_authors = ", ".join(project_authors)

print(f"The people who worked on this project are: {project_a
uthors}.")
```

The output is now exactly what we want:

```
The people who worked on this project are: Mike, Sofia, Hele
n.
```

We could achieve the same thing by putting the `join` call right inside the curly braces of the f-string:

```
project_authors = ["Mike", "Sofia", "Helen"]

print(f"The people who worked on this project are: {', '.join
(project_authors)}.")
```

Just be careful with your quotes! If you use the same quotes to define the join string, you're going to end up cutting off the outer string early.

One thing to be aware of when using `join` is that we can only join collections of strings. If we have a list of numbers, we have to convert each number to a string first.

- We start with a list of numbers, called `numbers`.
- Then we create an empty list, `stringified_numbers`. This will hold our numbers converted to strings.
- Using a `for` loop, we iterate over the `numbers` list. We convert each number to a string and append it to `stringified_numbers`.
- Finally, we use `join` with `stringified_numbers`.

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

stringified_numbers = []

for number in numbers:
        stringified_numbers.append(str(number))

print(', '.join(stringified_numbers)) # 1, 2, 3, 4, 5
```

I appreciate that this looks really cumbersome, but there are much more efficient and succinct ways of doing something like this that we'll explore later in the series.

Splitting up a string

Very often, instead of combining a collection into a single string, we want to convert a string into another collection of several items. For example, we might want to do this when processing user input if the user provides multiple values in response to a single prompt.

One common way of doing this is to use the `split` method. When calling `split`, we often need to pass in a second string that we'll use as a separator sequence. Whenever this sequence of characters is

as a separator sequence. Whenever this sequence of characters is encountered, that marks the end of one value, and the start of the next.

For example, let's say we want to get five numbers from the user, and we request that these numbers are comma separated. We might write something like this:

```
user_numbers = input("Please enter 5 numbers separated by com
mas: ") # 1,2,3,4,5
numbers_list = user_numbers.split(",")

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

Here we specified a comma as the separator sequence, so each comma marks the end of a value we want to add to the collection.

We could ask the user instead to enter numbers separated by dashes, and then we would have to pass in a dash to `split` as the separator:

```
user_numbers = input("Please enter 5 numbers separated by das
hes: ") # 1-2-3-4-5
numbers_tuple = user_numbers.split("-")

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

What we get back from `split` is a list, but we can always create another type of collection by passing the result to `tuple` , for example, if that's what we want:

```
user_numbers = input("Please enter 5 numbers separated by com
mas: ") # 1,2,3,4,5
numbers_tuple = tuple(user_numbers.split(","))

print(numbers_tuple) # ('1', '2', '3', '4', '5')
```

One thing to be aware of is that `split` won't strip out any whitespace

for us. If the user writes the numbers with commas followed by spaces, those spaces end up in the strings we put in the new collection:

```
user_numbers = input("Please enter 5 numbers separated by com
mas: ") # 1, 2, 3, 4, 5
numbers_list = user_numbers.split(",")

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

Sometimes this isn't really an issue, but sometimes you may need to go through the collection with a `for` loop and clear things up using `strip`.

```
user_numbers = input("Please enter 5 numbers separated by com
mas: ") # 1, 2, 3, 4, 5
user_numbers = user_numbers.split(",")

numbers_list = []

for number in user_numbers:
        numbers_list.append(number.strip())

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

You can use the same kind of approach to process the individual strings in other ways as well.

If we don't specify a separator sequence for `split`, it's going to split based on any length of whitespace. This is quite handy, because we know that whatever we get back, we're not going to have to bother stripping it. The user could enter something like this:

```
1 2 3 4 5
```

And we'd still get a list like this:

```
['1', '2', '3', '4', '5']
```

This also works if we're working with strings split across different lines! More on that in a moment.

We don't always need to call `split`. If we just want to put every character as a different item in a list or tuple, we can just pass the string to the `list` or `tuple` function instead:

```
sample_string = "Python"

print(list(sample_string)) # ['P', 'y', 't', 'h', 'o', 'n']
print(tuple(sample_string)) # ('P', 'y', 't', 'h', 'o', 'n')
```

The newline character

One thing you may be wondering is, how do we tell the computer that we should have a line break?

There's actually an invisible character that gets added when we mark the end of a line by pressing "Enter". If you go into a word processor, and you press "Enter" to go to the next line, what happens when you press "Backspace"? We delete something, and that gets us back to the previous line. What you've deleted is the newline character which marks the end of the line.

In Python, we represent this character as `\n`, and we can use it in strings just like any other character. For example, I can write a string like this:

```
print("Super Special Mega Awesome Program\n\nBy Phillip Best"
)
```

Here I've added two line breaks after the title of some program, and then information about the program author. The output will look like

this:

```
Super Special Mega Awesome Program

By Phillip Best
```

If we want to, we can join strings using this character, and it's possible to split strings using `\n` as well, but this isn't really useful for user input. It can, however, be very useful when working with files, which is something we're going to cover next week.

Slicing

So far we've looked at breaking apart strings, and joining the values of other collections, but what if we just want to grab some *part* of a collection? For this, we can use slicing.

Slicing is a new way for us to use a subscription expression with sequences. Instead of providing a single index, we specify a range of indices. What we'll get back is a collection of the same type that we sliced, containing all the items in the specified range. One great thing about slices is that they give us a new collection, leaving the original untouched.

Let's look at an example with a string. Let's say we have the string `Python`, and I want the first 3 characters of this string for some reason. Using a slice, we'd write this:

```
original_string = "Python"
sliced_string = original_string[0:3]

print(sliced_string)  # Pyt
```

Here our slice was written as `[0:3]`, which means we want to start at index `0`, and we want to grab every item up to, but not including index

`3`. What we get back is a new string containing just three characters: `"Pyt"`.

There's another way we could have written this slice, which is like this: `[:3]`. If we don't specify a value for the start index, the slice starts at the beginning of the sequence. If don't specify a stop index, the slice stops at the end of the sequence.

So if we wrote this `[3:]`, this means give me everything from index `3` onward. We can see an example here:

```
original_string = "Python"
sliced_string = original_string[3:]

print(sliced_string)  # hon
```

Back in day 4, we saw that we can use negative indices to access elements in a sequence, and we can use negative indices with slices as well. However, it's really important that you understand that `[3:-1]` isn't the same thing as `[3:]`, because with slices, the stop index isn't inclusive.

When we write `[3:-1]`, we're saying we want everything from index `3` up to, but not including, the final element. When we write `[3:]`, we're saying we want everything from index `3` onward. We can see the difference here:

```
original_string = "Python"

print(original_string[3:])  # hon
print(original_string[3:-1])  # ho
```

If you want to copy an entire collection for some reason, we can actually take a slice containing the whole sequence using `[:]`.

Slices are actually an incredibly versatile tool, and there's a lot more to learn. If you're interested, there are some additional resources at the

learn. If you're interested, there are some additional resources at the end of this post, which talk about slices in a lot more detail.

The `len` function

If we want to find out how many items are in a given collection, we can use the `len` function, short of "length".

This will work for strings, tuples, lists, and any many of the other types we're going to look at later in the series. If you try to call `len` on a type which doesn't support this operation, you'll get an exception like this:

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    len(12)
TypeError: object of type 'int' has no len()
```

Assuming pass an appropriate type to `len` , it will return an integer representing the number of items in the collection:

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

Exercises

1) Ask the user to enter their given name and surname in response to a single prompt. Use `split` to extract the names, and then assign each name to a different variable. For this exercise, you can assume that the user has a single given name and a single surname.

2) Print the list, `[1, 2, 3, 4, 5]` , in the format `1 | 2 | 3 | 4 | 5` using the `join` method. Remember that you can only join collections of strings, so you're going to need to do some initial processing of the list of numbers.

3) Below you'll find a short list of quotes:

```
 quotes = [
        "'What a waste my life would be without all the beaut
iful mistakes I've made.'",
        "'A bend in the road is not the end of the road... Un
less you fail to make the turn.'",
        "'The very essence of romance is uncertainty.'",
        "'We are not here to do what has already been done.'"
 ]
```

Each quote is a string, but each string actually contains quote characters at the start and end. Using slicing, extract the text from each string, without these extra quote marks, and print each quote.

You may also want to try a solution using strip.

4) Ask the user to enter a word, and then print out the length of the word. You should account for any excess whitespace in the user's input, so you're going to have to process the string before you find its length.

If you want to take this a little bit further, you an ask the user for a long piece of text. You can then tell them how many how many characters are in the text overall, and you can also provide them a word count.

You can find our solutions to the exercises here.

Project

Once you've finished the exercises above, you should try the end of week project where we'll be dealing with high budget movies.

Additional Resources

If you want to learn more about slicing, we have three posts you can look at.

The first two are a mini-series that cover everything you need to know about slicing in Python: Part 1, Part 2.

The third post is about how to use slicing to create a new sequence where the order of the items is reversed. I'd recommend you read through the first two posts before reading this one.
