Debugging Lists

by Sophia

WHAT'S COVERED

In this lesson, you will learn about some of the common issues with lists and how to debug them. Specifically, this lesson covers:

1. Debugging Lists

As we discussed earlier, iterable objects such as lists, dictionaries, sets, and tuples are known generically as data collection types. Since these data collection types can contain any object, we can implement different data collection types into other ones, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. When we covered multiple dimensions, we only used lists within lists.

Compound data collections are useful, but they are prone to errors caused when a data collection has the wrong type, size, or composition, or perhaps we wrote some code and forgot the types for our data and introduced an error. For example, if we are expecting a list with one integer and we get a plain old integer (not in a list), it won't work.

Careless use of lists (and other mutable objects) can lead to long hours of debugging. We'll cover some common pitfalls and ways to avoid them. It's important to remember that most list methods modify the arguments that are being passed into the method and return None as the return type. The type **None** is a return type that simply means that there is no data that is returned. This is the opposite of the string methods, which return a new string and leave the original alone.

For example, consider if we are used to writing string code like this:

EXAMPLE

```
myString = " Fun things for Us "
myString = myString.strip()
print(myString)
```

Note that the **.strip() method** is a string method that removes any spaces before the first character or after the last character. Did you notice that `myString` has spaces before and after the end of the string? Let's run it and see what the output is.

```
Fun things for Us
```

As expected, the `.strip()` method removed all the spaces before and after the string. This is useful if we have any extra spaces in the text before and after the string, especially if we need to concatenate different strings together.

TRY IT

Directions: Go ahead and try using the `.strip()` method in the IDE on a string of your choice. No matter the amount of spaces before and after your string, your output should see them removed.

Here is a common mistake when using a list. It is tempting to write code for the list like this:

EXAMPLE

```
myList = ["ice cream","frozen yogurt","sorbet"]
myList = myList.sort()
print(myList)
```

So we want to sort the list using the `.sort()` method. Let's see the output.

```
None
```

Oops, that was completely wrong as the result showed the type None rather than the list in sorted order. Because the sort returns None, the next operation you perform with `myList` is likely to fail.

BIG IDEA

Before using list methods and operators, you should make sure you review the list methods we have discussed so far. Another great resource on using methods is www.python.org/, the actual website of the Python programming language. For some additional information on using built-in methods and functions, visit their library index at docs.python.org/3/library/index.html. This index has links to built-in functions and methods. Since methods are associated with the object they are used, search the index for your object (for example, a "list") and then look for the method on the accompanying pages. It is always a good idea to understand what a method will do to an object before you write too much code. It's always better to test first and then continue.

For our sorting issue, we should have used the `.sort()` method like this:

EXAMPLE

```
myList = ["ice cream","frozen yogurt","sorbet"]
myList.sort()
print(myList)
```

The output looks like:

```
['frozen yogurt', 'ice cream', 'sorbet']
```

That looks better! Our list is now sorted alphabetically based on frozen desserts.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use the `.pop()` and `.remove()` methods or the `del` statement. You can even use the `slice` operator. To add an element, you can use the `.append()` method or the `+` operator. But don't forget that these are the correct methods or ways to add/remove elements given certain conditions.

Let's cover some of these conditions in both correct and incorrect instances.

TRY IT

Directions: Go ahead and follow the instructions below to test some correct use conditions.

We will start with the following lines of code. Enter these lines of code in the IDE.

```
myFirstList = ["ice cream","frozen yogurt","sorbet"]
myElement = "gelato"
#each test line goes here…
print(myFirstList)
```

Now for the correct conditions. Try each of the following lines of code by replacing the commented line with these, one at a time.

```
myFirstList.append(myElement)

myFirstList = myFirstList + [myElement]
```

Did you get the "gelato" added to the list? You should have seen the output below when using the `.append()` method or `+` operator.

```
['ice cream', 'frozen yogurt', 'sorbet', 'gelato']
```

Now let's try some incorrect use cases.

TRY IT

Directions: Go ahead and follow the instructions below to test some incorrect use conditions. We will discuss each condition after its use.

Again, we will start with the original lines of code.

```
myFirstList = ["ice cream","frozen yogurt","sorbet"]
```

```
myElement = "gelato"
#each test line goes here…
print(myFirstList)
```

Try each of the following lines of code by replacing the commented line with these, one at a time.

```
myFirstList.append([myElement])
```

So what did that do?
The line above isn't correct as it attempts to add a list within the list rather than just the element. The added [] around the element is what indicates that it is a list.

Try this next line of code, replacing the last line.

```
myFirstList = myFirstList.append(myElement)
```

So what did that do?
This is incorrect as the `.append()` method returns None. Meaning that after we append the item to the list, None is being returned and set to `myFirstList`, removing all items.

Ok, next line of code, replace the line and run it.

```
myFirstList + [myElement]
```

So what did that do?
Did you see the original list? This is incorrect as we cannot add to the list in this way. We must use the `.append()` method instead.

For the final one, add this line of code, replace and run it.

```
myFirstList = myFirstList + myElement
```

So what did that do?
Similar to the previous line, we have to use the `.append()` method to add items to a list.

Notice that only the last incorrect condition produced a runtime error; the first three cases do not produce an error, but they do the wrong thing.

Remember that there are many ways to do operations with lists. A big part of using lists (or any other iterable objects)—or really any operations, for that matter—in Python is to know what can be used with what. Even seasoned Python programmers will keep references like the Python.org library index handy. And again, it is always best practice to test early and often to make sure your program is doing what you are expecting.

Do you wish to keep your original list?
If you want to use a method like `.sort()` that modifies your original list, but you want to keep the original list as well, you can make a copy. In the following example code, `myFirstList` remains the same while the `anotherList` list contains the sorted content.

TRY IT

Directions: Try the code below to get a better understanding on how to copy a list while keeping the original list intact.

```
myFirstList = ["ice cream","frozen yogurt","sorbet"]
anotherList = myFirstList[:]
anotherList.sort()
print(anotherList)
print(myFirstList)
```

The output shows:

```
['frozen yogurt', 'ice cream', 'sorbet']
['ice cream', 'frozen yogurt', 'sorbet']
```

As you can see, the original list (`myFirstList`) is intact and still in the order that we added the elements, but the copied list (`anotherList`) has been sorted. Note, we used the `slice` operator and only included the colon (:), which means that we're using all elements of the list since we have not included either the first or last indexes. Remember, with the `slice` operator, if you only leave the colon and no indexes, the slice will just be a copy of a whole list.

As you work with bigger data collection types that have more elements it can become unwieldy to debug by printing and checking the data by hand. Here are some suggestions for debugging large data collection types:

- Scale down the input - If possible, reduce the size of the data collection type. For example, if the program reads input from the user one line at a time, start with just the first 10 lines, or with the smallest example, you can find. You can either edit the files themselves or (better) modify the program so it reads only the first few line numbers. If there is an error, you can reduce the number of lines being read to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.
- Check the count of elements and types - Instead of printing and checking the entire data collection type, consider printing summaries of the data: for example, the number of elements in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value through a check on the data type using the `type()` function.
- Write self-checks - Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest.
- Pretty print the output - Formatting debugging output can make it easier to spot an error. For example, having the exception handling on different lines instead of all exceptions on a single line. Part of this is taking the time to change the exception messages to something that is readable to any user. Imagine getting a computer error message that says "ERROR: X65489746" rather than seeing an error message that says, "The input that was entered was expected to be a number between 1-100, please re-enter the value."

After reading these suggestions, remember, that the more time you spend building your code piece by piece, the more you can reduce the time you spend debugging.

TERMS TO KNOW

None

The type None is a return type that simply means that there is no data that's returned.

strip()

The `.strip()` method is a string method that removes any spaces before the first character or after the last character.

SUMMARY

In this lesson, we learned that iterable objects such as lists, dictionaries, sets, and tuples (also called data collection types) can contain data from each other. These compound data collections are useful but can be prone to errors if the programmer is not careful with method usage on particular objects. It is always ideal to review which methods work with which objects while coding. We **debugged lists** while testing some correct and incorrect code. Finally, we covered some suggestions and ways to approach debugging, especially with larger collections of data where manually checking by hand may not be possible.

Best of luck in your learning!

TERMS TO KNOW

.strip()

The .strip() method is a string method that removes any spaces before the first character or after the last character.

None

The type None is a return type that simply means that there is no data that's returned.