Multiple Dimensions

by Sophia

WHAT'S COVERED

In this lesson, you will learn about multiple dimensions in lists. Specifically, this lesson covers:

1. Multiple Dimensions

Before we discuss multiple dimensions in Python, let's identify what multiple dimensions truly look like first.

One Dimension

We are very familiar with a dimension of one (one dimension). It resembles what we have seen with list data collection types so far. Basically, it's a straight line on a flat plane.

A one-dimensional listing would look like this on paper.

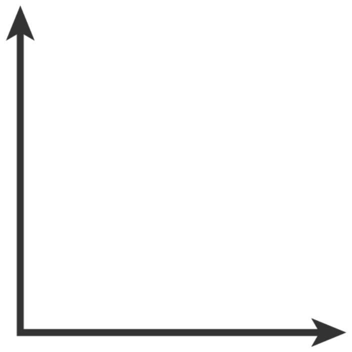
element	sun	fun	run	bun	nun	pun
Index value	0	1	2	3	4	5

The index position is one integer, so if we wanted to locate "run" from this listing, we can just use the single index position. Remember that list elements start at 0.

"run" is at index value 2.

Two Dimension

Ok, so how different is a two-dimensional collection of data? We have discussed two-dimensional lists previously. It was quick, but if you remember when we discussed nesting a list within a list, well, that's adding a second dimension. A nested list is a two-dimensional list or 2D for short. What does that look like graphically?



You likely have seen this in high school. It’s basically an X/Y chart. It has two dimensions, one running horizontally and the other vertically, still on a flat plane.

A two-dimensional listing would look like this on paper.

X/Y indexes		X →		
		0	1	2
Y ↓	0	fun	sun	run
	1	bun	nun	pun
	2	fan	van	man
	3	can	tan	ran

Now there are two index positions for each element. If we wanted to locate "fan" from this two-dimensional listing we would need to identify both index positions, starting with the column and then the row.

"fan" is at position 0,2 (X,Y or Column/Row)

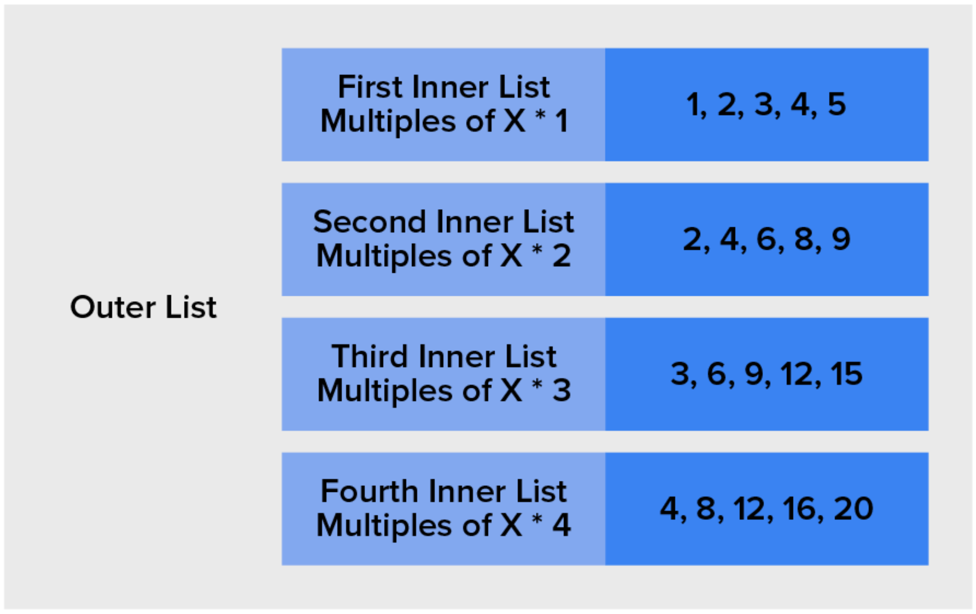
Three Dimension

Once we break into three-dimensional lists or other data collection types, things get even more complicated. Now we are looking at X, Y, and Z index positions. We are past anything that is easy to see on “paper” or a flat plane. We are not going to discuss any more than two dimensions in this course, since utilizing three-dimensional collections of data is rarely used unless it is for 3D object modeling or mapping.

Now back to Python. As you saw above, there can be more than one dimension when it comes to a list (or other data collection type) in Python. Think about how a teacher stores their grades for a class. They have a list of students and within that list, they have a list of assignments. For each of those assignments, they’ll assign a grade. It’s important to remember that a list can hold other lists and that the basic principles can be applied multiple times. Multi-dimensional lists are just that: lists within a list. Any data collection type can contain other collection types as well. Let’s take a look at an example of a nested list. In this case, we have a **two-dimensional** list (2D list for short) with four lists within the first list (see outer list in the graphic).

Here is a 4 X 5 list example:

© 2025 SOPHIA Learning, LLC. SOPHIA is a registered trademark of SOPHIA Learning, LLC.



The first inner list has multiples of 1 up to 5 numbers (1 * 1, 1 * 2, and so on.). The second inner list has multiples of 2 up to 5 numbers, and so forth.

Every 2D list is a nested list, but not all nested lists are 2D lists. A 2D list has the same number of elements per inner (nested) list. With inner lists, they could have a varying number of elements, whereas a 2D list requires the same number of elements for each inner list.

Let's see this nested 2D list in Python code.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
print(multiplesList)
```

2D list in the output:

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Right now this 2D list is a long row of elements separated by the square brackets.
We can access the first list by using the index.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
print(multiplesList[0])
```

The output of the first nested list.

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

We output just the first list from the full 2D listing.

If we wanted to access a specific element within that first list, we can again use the square brackets after the first set. If we wanted the third element, we would use the 2 as the index. For visualization, notice that we'll split up the list onto multiple lines.

Note: In the example below and using the IDE, we just selected the Enter key after each comma and then hit the Tab key until the inner lists were vertically aligned on the screen.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5],
[2, 4, 6, 8, 10],
[3, 6, 9, 12, 15],
[4, 8, 12, 16, 20]]
print(multiplesList[0][2])
```

The output shows 3.

```
3
```

Looks correct! We located and output number 3.

TRY IT

Directions: Now you try to output a few elements in this 2D list. Enter the list below into the IDE and we'll ask you to locate and `print()` some elements.

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Use the + button to the right of each element request to see if you nailed the code to output it correctly.

Print to the screen the number 12 the first time you see it. +

print(multiplesList[2][3])

Print to the screen number 12 the second time you see it in the list. +

print(multiplesList[3][2])

.append() Method

We can add another nested inner list to the 2D array by using the .append() method. Let's add a fifth nested list that is multiples of 5.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
multiplesList.append([5, 10, 15, 20, 25])
print(multiplesList)
```

The output looks like:

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]
```

© 2025 SOPHIA Learning, LLC. SOPHIA is a registered trademark of SOPHIA Learning, LLC.

Now we have five nested lists in the original list.

If we wanted to append an element to an inner list (a list within the outer list), we can use the `.append()` method and an index value like the following:

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
multiplesList[0].append(6)
print(multiplesList)
```

The output looks like:

```
[[1, 2, 3, 4, 5, 6], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Notice that this adds the 6 to the first nested list since we used the index 0.

Note: We can't use the `.append()` method if we have multiple elements to add to a nested inner list, as it would instead do one of two things:

1. We would get a TypeError if we tried to pass more than one element using the `.append()` method, since the `.append()` method takes only one argument at a time.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
multiplesList[0].append(6, 7, 8)
print(multiplesList)
```

The output shows TypeError.

```
Trackback (most recent call last):
    File "/home/main.py", line 2, in <module>
     multiplesList[0].append(6, 7, 8)
TypeError: list.append() takes exactly one argument (3 given)
```

The TypeError is saying we tried to pass 3 arguments at a time.

2. If we tried to pass these multiple elements using the `.append()` method and list brackets, we would actually get a nested inner list within our original first nested inner list. This would be the 6th element of the outer list: outer list = 1, original inner nested lists = 4, now an inner nested list (within the first of the original inner nested list) = 1,... total = 6. This would move into a three-dimensional list and things would really start to get complex.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
multiplesList[0].append([6, 7, 8])
print(multiplesList)
```

The output shows:

```
[[1, 2, 3, 4, 5, [6, 7, 8]], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

See, the 6, 7, and 8 now appear as an inner nested list within the first of the original inner nested lists, and that was not what we intended to do.

.extend() Method

Instead, we want to use the `.extend()` method (another method that we used in a previous lesson with lists) which would just add the elements to the first original inner nested list.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
multiplesList[0].extend([6, 7, 8])
print(multiplesList)
```

The output is as follows:

```
[[1, 2, 3, 4, 5, 6, 7, 8]], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Now we added 6, 7, and 8 to the first original inner nested list!

Can we move into even more dimensions?
Sure, we can extend that past example even further if we wanted to have more than two dimensions and have additional lists within a list, but that can get quite difficult to visualize. For example, this could represent a 3x3x3 cube with the value of each row having the same numeric value.

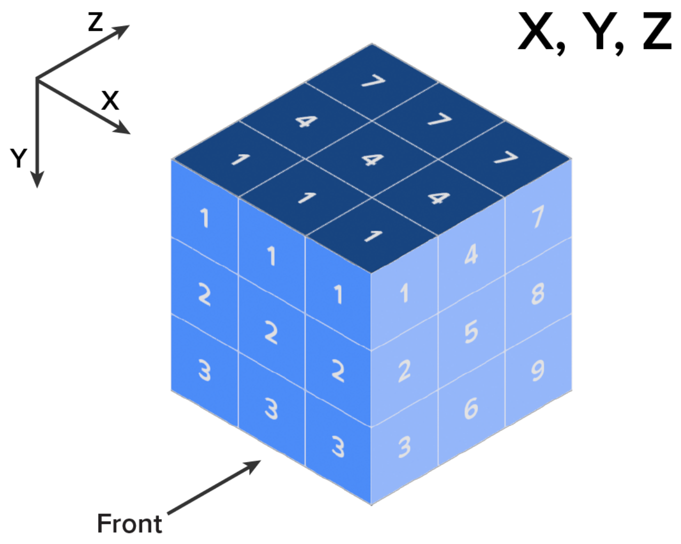
EXAMPLE

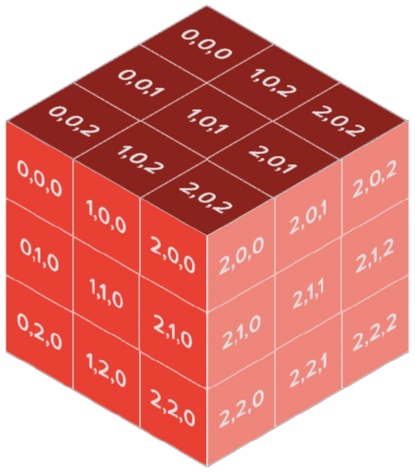
```
multiplesList = [[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[4, 4, 4],[5, 5, 5],[6, 6, 6]], [[7, 7, 7],
[8, 8, 8],[9, 9, 9]]]
print(multiplesList)
```

The output shows a 3D list.

```
[[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[4, 4, 4],[5, 5, 5],[6, 6, 6]], [[7, 7, 7], [8, 8, 8],[9, 9, 9]]]
```

Again, we are not going to go into too much detail about using multiple dimensions above 2D. On paper, the easiest way to "see" this three-dimension list would be to see it as a Rubik's cube: 3X3X3. Each row would be the same numeric value. It would have the X columns and Y rows, but now we would also have Z columns for "depth" to this cube.





So, if we wanted to get an element from a certain position we would need to provide the X, Y, and Z positions to retrieve the element. As you can see with the multiple indexes, coding can become very complex.

TERM TO KNOW

2D (Two Dimensions)
A nested list or other data collection type (iterable objects like lists, set, tuple, or dictionary) inside another iterable object is called two-dimension or 2D for short. Any data collection type can contain other nested collection types as well in multiple dimensions.

SUMMARY

In this lesson, we learned more about using lists with **multiple dimensions**. We discussed the difference between one, two, and more than two dimensions, although we will not go into more detail on using more than two dimensions (2D) in this course. Finally, we brushed up on the ways to interact with the multiple dimension lists using the .append() and .extent() methods that we learned in an earlier lesson.

© 2025 SOPHIA Learning, LLC. SOPHIA is a registered trademark of SOPHIA Learning, LLC.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT **www.py4e.com/html3/** LICENSE: **CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED**.

TERMS TO KNOW

2D (Two Dimensions)

A nested list or other data collection type (iterable objects like lists, set, tuple, or dictionary) inside another iterable object is called two dimension or 2D for short. Any data collection type can contain other nested collection types as well in multiple dimensions.