Lesson 26. Advanced operations with lists

After reading lesson 26, you'll be able to

- Build lists whose elements are lists
- Sort and reverse list elements
- Convert a string into a list by splitting on a character

A list is typically used to represent a collection of items, frequently but not necessarily of the same type. You'll see that it may be useful for the list elements to be lists themselves. For example, suppose you want to keep a list of all the items in your house. Because you have many items, it'll be more organized to have sublists, where each sublist represents a room, and a sublist's elements are all the items in that room.

At this point, it's important to take a step back and understand what has been going on with this new mutable object, a list. Lists are directly modified by any actions you do on them. Because the list is directly modified, you don't reassign the list to a new variable after an operation; the list itself now contains the changed values. To see the value of the modified list, you can print it.

Consider this

Your friend can recite the number pi up to 100 digits. You add each digit into a list as he tells it to you. You want to figure out how many zeros are in the first 100 digits. How can you quickly do this?

Answer:

If you sort the list, you can count the zeros at the beginning of the list.

26.1. SORTING AND REVERSING LISTS

After you have a list of elements, you can perform operations that rearrange elements in the whole list. For example, if you have a list of students in your class, you don't need to keep two lists of the same students: one sorted and one unsorted. You can start out with an unsorted list and then sort it directly when needed. When you care only about the contents of the list, storing it in a sorted manner may be preferred. But note that after it's sorted, you can't go back to the unsorted version of the list unless you re-create it from scratch.

Because lists are mutable, you can sort a list, using the operation `sort()`, so that the list elements of the original list are now in sorted order. The command `L.sort()` will sort the list `L` in ascending order (for numbers) and lexicographically (for letters or strings). In contrast, if you wanted to sort items in an immutable tuple object, you'd be creating many intermediary objects as you're concatenating the items from end to beginning (take the last item and put it at index 0, take the second-to-last item and put it at index 1, and so on).

Reversing a list may also be useful. For example, if you have a list of the names of your students and you sorted them alphabetically, you can reverse the list and have them sorted in reverse alphabetical order. The command `L.reverse()` reverses the list `L` so that the element at the front is now at the end, and so on.

Listing 26.1. Sorting and reversing a list

```
heights = [1.4, 1.3, 1.5, 2, 1.4, 1.5, 1]
heights.reverse()                              1
print(heights)                                 2
heights.sort()                                 3
print(heights)                                 4
```

- ***1*** **Reverses original list**
- ***2*** **Prints [1, 1.5, 1.4, 2, 1.5, 1.3, 1.4] because the preceding line reverses the original list by moving the first element to the last position, the second element to the second-to-last position, and so on**
- ***3*** **Sorts list in ascending order**
- ***4*** **Prints [1, 1.3, 1.4, 1.4, 1.5, 1.5, 2] because the preceding line sorted the list in ascending order**
- ***5*** **Reverses sorted list**
- ***6*** **Prints [2, 1.5, 1.5, 1.4, 1.4, 1.3, 1] because the preceding line reversed the list sorted in ascending order**

Quick check 26.1

Q1:

What's the value of list L after each operation?

```
L = ["p", "r", "o", "g", "r", "a", "m", "m", "i",
L.reverse()
L.sort()
L.reverse()
L.reverse()
L.sort()
```

You've seen lists whose elements are floats, integers, or strings. But a list can contain elements of any type, including other lists!

26.2. LISTS OF LISTS

If you want to program a game, especially a game that relies on the user being in a certain position, you'll often want to think about the position on a board represented in a two-dimensional coordinate plane. Lists can be used to help you represent the two-dimensional coordinate plane by using one list whose elements are also lists. The following listing creates a list L whose three elements are empty lists, and then populates the list with elements.

Listing 26.2. Creating and populating a list of lists

```
L = [[], [], []]                1
L[0] = [1,2,3]                  2
L[1].append('t')                3
L[1].append('o')                4
L[1][0] = 'd'                   5
```

- ***1*** **Empty list of lists**
- ***2*** **L has the value [[1,2,3], [], []] because you set the element at index 0 to be the list [1,2,3].**
- ***3*** **L has the value [[1,2,3], ['t'], []] because you appended the string 't' to the middle empty list.**
- ***4*** **L has the value [[1,2,3], ['t', 'o'], []] because you appended the string 'o' to the already mutated middle list.**
- ***5*** **L has the value [[1,2,3], ['d', 'o'], []] because you accessed the element at index 1 (a list) and then accessed that object's element at index 0 (letter t) to change it (to the letter d).**

Working with lists of lists adds another layer of indirection when indexing into the list to work with its elements. The first time you index a list of lists (or even a list of lists of lists of lists), you access the object at that position. If the object at that position is a list, you can index into that list as well, and so on.

You can represent a tic-tac-toe board with a list of lists. Listing 26.3 shows the code for setting up the board with lists. Because lists are one-dimensional, you can consider each element of the outer list to be a row in the board. Each sublist will contain all the elements for each column in that row.

Listing 26.3. Tic-tac-toe board with lists of lists

```
x = 'x'
o = 'o'
empty = '_'

board = [[x, empty, o], [empty, x, o], [x, empty, emp
```

- ***1*** **Variable x**
- ***2*** **Variable o**

- **4 Replaces every variable with its value. Variable board has three rows (one for each sublist) and three columns (each sublist has three elements).**

This tic-tac-toe board represented in code looks like this:

```
x _ o
_ x o
x _ _
```

Using lists inside lists, you can represent any size tic-tac-toe board by adjusting the number of sublists you have and the number of elements each sublist contains.

Quick check 26.2

Using the variables set up in listing 26.3, write the line of code to set up a board that looks like this:

1

A 3 × 3 board

_ _ _

x x x

o o o

2

A 3 × 4 board

x o x o

o o x x

o _ x x

26.3. CONVERTING A STRING TO A LIST

Suppose you're given a string that contains email data separated by commas. You'd like to separate out each email address and keep each address in a list. The following sample string shows how the input data might look:

```
emails = "zebra@zoo.com,red@colors.com,tom.sawyer@boo
```

You could solve this problem by using string manipulations, but in a somewhat tedious way. First, you find the index of the first comma. Then you save the email as the substring from the beginning of the string `emails` until that index. Then you save the rest of the string from that index until the end of `emails` in another variable. And finally, you repeat the process until you don't have any more commas left to find. This solution uses a loop and forces you to create unnecessary variables.

Using lists provides a simple, one-line solution to this problem. With the preceding `emails` string, you can do this:

```
emails_list = emails.split(',')
```

This line uses the operation `split()` on the string named `emails`. In the parentheses to `split()`, you can put the element on which you'd like to split the string. In this case, you want to split on the comma. The result from running that command is that `emails_list` is a list of strings that contains every substring between the commas, as shown here:

```
['zebra@zoo.com', 'red@colors.com', 'tom.sawyer@book.
```

Notice that each email is now a separate element in the list `emails_list`, making it easy to work with.

Quick check 26.3

Write a line of code to achieve the following tasks:

1

Split the string `" abcdefghijklmnopqrstuvwxyz"` by the space

2

Split the string `"spaces and more spaces"` by words.

3

Split the string `"the secret of life is 42"` on the letter `s`.

With the operations you saw on lists in lesson 25 (sorting and reversing a list), you're now able to simulate real-life phenomena: stacks and queues of items.

26.4. APPLICATIONS OF LISTS

Why would you need to simulate a stack or a queue by using a list? This is a somewhat philosophical question, and it hints at what you'll see in the next unit. A more basic question is why do I need a list object when I can just create a bunch of integer/float/ string objects and remember the order I want them in? The idea is that you use simpler objects to create more complex objects that have more specific behaviors. In the same way that a list is made up of an ordered group of objects, a stack or a queue is made up of a list. You can make up your own stack or queue object so that their construction is the same (you use a list) but their behavior is different.

26.4.1. Stacks

Think of a stack of pancakes. As they're being made, new pancakes are added to the top of the stack. When a pancake is eaten, it's taken from the top of the stack. You can mimic this behavior with a list. The top of the stack is the end of the list. Every time you have a new element, you add it to the end of the list with `append()`. Every time you want to take an element out, you remove it from the end of the list with `pop()`.

Listing 26.4 shows an implementation of a pancake stack in Python. Suppose you have blueberry and chocolate pancakes. A blueberry pancake is represented by the element `'b'` (letter *b* as a string) and a chocolate pancake as `'c'` (letter *c* as a string). Your pancake stack is originally an empty list (no pancakes made yet). One cook makes batches of pancakes; the cook is also a list with pancake elements. As soon as a batch is made by the cook, the batch is added to the stack by using `extend()`. Someone eating a pancake can be represented by using the `pop()` operation on the stack.

Listing 26.4. A stack of pancakes represented with a list

```
stack = []
cook = ['blueberry', 'blueberry', 'blueberry']
stack.extend(cook)
stack.pop()
stack.pop()
cook = ['chocolate', 'chocolate']
stack.extend(cook)
stack.pop()
cook = ['blueberry', 'blueberry']
stack.extend(cook)
stack.pop()
stack.pop()
stack.pop()
```

- ***1* Empty list**
- ***2* List of three pancakes made**
- ***3* Adds cook's pancakes to stack**
- ***4* Removes last element in list**
- ***5* New batch of pancakes**
- ***6* Adds cook's batch to stack at the end**
- ***7* Adds cook's batch to stack at the end**
- ***8* Removes last element in list**

Stacks are a *first-in-last-out* structure because the first item added to the stack is the last one taken out. Queues, on the other hand, are *first-in-first-out* because the first item added to a queue is the first one taken out.

26.4.2. Queues

Think of a grocery store queue. When a new person arrives, they stand at the end of the line. As people are being helped, the ones that have been in the queue the longest (at the front of the line) are going to be helped next.

You can simulate a queue by using a list. As you get new elements, you add to the end of the list. When you want to take out an element, you remove the one at the beginning of the list.

Listing 26.5 shows an example of a simulated queue in code. Your grocery store has one line, represented by a list. As customers come in, you use ap-

Listing 26.5. A queue of people represented by a list

```
line = []                       1
line.append('Ana')              2
line.append('Bob')              3
line.pop(0)                     4
line.append('Claire')           5
line.append('Dave')             5
line.pop(0)                     6
line.pop(0)                     6
line.pop(0)                     6
```

- ***1* Empty list**
- ***2* List of one person now in queue**
- ***3* List of two people now in queue**
- ***4* First person removed from the queue**
- ***5* New people added to the end of list**
- ***6* People removed from beginning of list**

Using more complex object types, such as lists, you can simulate real-life actions. In this case, you can use specific sequences of operations to simulate stacks and queues of objects.

Quick check 26.4

Are the following situations best representative of a queue, a stack, or neither?

1

The Undo mechanism in your text editor

2

Putting tennis balls in a container and then taking them out

3

Cars in a line waiting for inspection

4

Airport luggage entering the carousel and being picked up by its owner

SUMMARY

In this lesson, my objective was to teach you more operations that you can do with lists. You sorted a list, reversed a list, created lists that contained other lists as elements, and converted a string into a list by splitting it on a character. Here are the major takeaways:

- Lists can contain elements that are other lists.
- You can sort or reverse a list's elements.
- Behaviors of stacks and queues can be implemented using lists.

Let's see if you got this...

Q26.1

Write a program that takes in a string containing city names separated by commas, and then prints a list of the city names in sorted order. You can start with this:

```
    cities = "san francisco,boston,chicago,indianapol
```

Q26.2

Write a function named `is_permutation`. It takes in two lists, `L1` and `L2`. The function returns `True` if `L1` and `L2` are permutations of each other. It returns `False` otherwise. Every element in `L1` is in `L2`, and vice versa, only arranged in a different order. For example,

- `is_permutation([1,2,3], [3,1,2])` returns `True`.
- `is_permutation([1,1,1,2], [1,2,1,1])` returns `True`.
- `is_permutation([1,2,3,1], [1,2,3])` returns `False`.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 25. Working with lists

NEXT
Lesson 27. Dictionaries as maps between objects