



## Lesson 25. Working with lists

After reading [lesson 25](#), you'll be able to

- Build Python lists
- Add items, remove items, and modify items in Python lists
- Perform operations on list elements

Mutable data types are types of objects on which you can perform operations, such that the object's value is modified; the modification is done in place, so to speak, so no copy of the object is made. There's a need for data types that are mutable, especially when working with large amounts of data; it's more efficient to modify data stored directly instead of copying it into a new object with every operation.

Instead of creating new objects, it's sometimes useful to reuse an object and modify its value. This is especially true when you have an object that represents an ordered collection of other objects. Most programming languages have a data type to represent this type of ordered collection that you can mutate. In Python, this is called a *list*. A list is a new type of object that you haven't seen before. It represents an ordered collection of other object types. Among many others, you can have lists of numbers, lists of strings, or even a list of a mix of object types.

Often you'll find that you need to store objects in a certain order. For example, if you keep a list of grocery items, the first item will be on the first line, the second on the next line, and so on. The fact that a list is ordered comes from the idea that each item has a specific place in the list, and that your first item will always be at the first place in the list, and the last item will be at the final place in the list. This lesson will show you how to create lists and perform operations to mutate the lists; these operations may move items around in the list, but a list will always have an item in its first place and in its final place. Dictionaries, which are covered in [lesson 27](#), are data types that store objects in an unordered fashion.

### Consider this

You maintain a list of all people who work at your company. The list is in a document on the computer. Which one of these events would require you to start a new document, copy over all information, and then act on the event?

- Someone joins the company.
- Someone leaves the company.
- Someone changes their name.
- The list will be sorted by first name, not last name.

Answer:

None.

### 25.1. LISTS VS. TUPLES

A list is a collection of any object type; it's like a tuple, which you've seen and worked with already. A tuple and a list both have an order to them, in that the first element in the collection is at index 0, the second element is at index 1, and so on. The main difference is that lists are mutable objects, whereas tuples aren't. You can add elements to, remove elements from, and modify elements in the same list object. With a tuple, every time you do an operation, you create a new tuple object whose value is the changed tuple.

Tuples are generally used to store data that's more or less fixed. Examples of data you'd store in a tuple are a pair of coordinate points in a 2D plane, or the page number and line number where a word occurs in a book. Lists are used to store data that's more dynamic. You use them when you're storing

sort, and so on. For example, you can use a list to store student grade scores or items in your fridge.

### Quick check 25.1

For each of the following, would you use a tuple or a list to store the information?

1

Pairs of letters and their position in the alphabet: (1, a), (2, b), and so on.

2

Shoe sizes of adults in the United States

3

Pairs of cities and their average snowfall from 1950 to 2015

4

Names of everyone in the United States

5

Names of apps on your phone/computer

### 25.2. CREATING LISTS AND GETTING ELEMENTS AT SPECIFIC POSITIONS

You create a Python list by using square brackets. The line `L = []` creates an object representing an empty list (a list with no elements), and binds the name `L` to that empty list object. `L` is a variable name, and you can use any variable name you want. You can also create a list with items initially in it. The following line creates a list of three items and binds the variable name `grocery` to that list:

```
grocery = ["milk", "eggs", "bread"]
```

As with strings and tuples, you can get the length of a list with `len()`. The command `len(L)`, where `L` is a list, tells you the number of elements that are in list `L`. The empty list has 0 elements. The length of the preceding list named `grocery` is 3.

Recall that in programming, we start counting from 0. As with strings and tuples, the first element in a list is at index 0, the second element is at index 1, and so on. The last element in the list is at `len(L) - 1`. If you have a grocery list `grocery = ["milk", "eggs", "bread"]`, you can get the value of each element by indexing into the list by using the square brackets, as with tuples and strings. The following code indexes into the list and prints each element:

```
grocery = ["milk", "eggs", "bread"]
print(grocery[0])
print(grocery[1])
print(grocery[2])
```

What happens if you try to index farther than the length of the list? Say you have the following code:

```
grocery = ["milk", "eggs", "bread"]
print(grocery[3])
```

You'll get this error, which tells you that you're trying to index into a list farther than the length of the list:

```
Traceback (most recent call last):
  File "<ipython-input-14-c90317837012>", line 2, in
    print(grocery[3])
IndexError: list index out of range
```

Recall that because the list contains only three elements and the first element is at index 0, then the last element in `grocery` is at index 2. An index position of 3 is beyond the limits of the list.

**Quick check 25.2**

You have the following list:

```
desk_items = ["stapler", "tape", "keyboard", "monitor"]
```

What's printed with each command?

**1**

```
print(desk_items[1])
```

**2**

```
print(desk_items[4])
```

**3**

```
print(desk_items[5])
```

**4**

```
print(desk_items[0])
```

**25.3. COUNTING AND GETTING POSITIONS OF ELEMENTS**

In addition to counting all elements in the list with the `len()` command, you can also count the number of times a particular element occurs by using the `count()` operation. The command `L.count(e)` tells you the number of times the element `e` occurs in the list `L`. For example, if you're looking at your grocery list, you could count the word *cheese* to make sure you have five kinds of cheeses on your list for your five-cheese pizza recipe.

You can also determine the index of the first element in the list that matches a value with the `index()` operation. The command `L.index(e)` tells you the index (starting from 0) of the first time the element `e` occurs in list `L`. The following listing shows how `count()` and `index()` work.

**Listing 25.1. Using count and index with a list**

```
years = [1984, 1986, 1988, 1988]
print(len(years))           1
print(years.count(1988))    2
print(years.count(2017))    3
print(years.index(1986))    4
print(years.index(1988))    5
```

- **1 Prints 4** because the list has four elements
- **2 Prints 2** because the list has the number 1988 twice
- **3 Prints 0** because 2017 doesn't occur at all in the list years
- **4 Prints 1** because 1986 occurs at index 1 (starting to count from 0)
- **5 Prints 2** because index finds only the index of the first occurrence of the number 1988

What happens if you try to get the index of an element that's not in the list? Say you have the following code:

```
L = []
L.index(0)
```

You'll get the following error when you run it. The error message contains information about the line number and line itself that leads to the error. The last line of the error message contains the reason the command failed: `ValueError: 0 is not in list`. This is the expected behavior of the `index` operation on a value that's not in the list:

```
Traceback (most recent call last):
  File "<ipython-input-15-b3f3f6d671a3>", line 2, in
    L.index(0)
ValueError: 0 is not in list
```

**Quick check 25.3**

**01:**

```
L = ["one", "three", "two", "three", "four", "three"]
print(L.count("one"))
print(L.count("three"))
print(L.count("zero"))
print(len(L))
print(L.index("two"))
print(L.index("zero"))
```

#### 25.4. ADDING ITEMS TO LISTS: APPEND, INSERT, AND EXTEND

An empty list isn't useful. Nor is a list that you have to populate as soon as you create it. After you create a list, you'll want to add more items to it. For example, given an empty piece of paper representing your weekly grocery list, you'll want to add items to the end of the list as you think of them instead of transcribing everything on a new piece of paper every time you want to add an item.

To add more elements to the list, you can use one of three operations: `append`, `insert`, and `extend`.

##### 25.4.1. Using `append`

`L.append(e)` adds an element `e` to the end of your list `L`. Appending to a list always tacks on the element to the end of the list, at the highest index. You can append only one element at a time. The list `L` is mutated to contain the one extra value.

Say you have an initially empty grocery list:

```
grocery = []
```

You can add an item as follows:

```
grocery.append("bread")
```

The list now contains one element in it: the string `"bread"`.

##### 25.4.2. Using `insert`

`L.insert(i, e)` adds element `e` at index `i` in list `L`. You can insert only one element at a time. To find the position for the insert, start counting elements in `L` from 0. When you insert, all elements at and after that index are shifted toward the end of the list. The list `L` is mutated to contain the one extra value.

Say you have this grocery list:

```
grocery = ["bread", "milk"]
```

You can insert an item between the two existing items:

```
grocery.insert(1, "eggs")
```

The list then contains all the items:

```
["bread", "eggs", "milk"]
```

##### 25.4.3. Using `extend`

`L.extend(M)` appends all elements from list `M` to the end of list `L`. You effectively append all the elements from list `M`, preserving their order, to the end of list `L`. The list `L` is mutated to contain all the elements in `M`. The list `M` remains unchanged.

Say you have an initial grocery list and a list of fun things to buy:

```
grocery = ["bread", "eggs", "milk"]
for_fun = ["drone", "vr glasses", "game console"]
```

You can extend the two lists:

```
grocery.extend(for_fun)
```

This gives you a master shopping list:

```
["bread", "eggs", "milk", "drone", "vr glasses", "game console"]
```

```

first3letters = []                                1
first3letters.append("a")                          2
first3letters.append("c")                          3
first3letters.insert(1,"b")                        4
print(first3letters)                              5
last3letters = ["x", "y", "z"]                     6
first3letters.extend(last3letters)                 7
print(first3letters)                              8
last3letters.extend(first3letters)                 9
print(last3letters)                             10

```

- **1** Empty list
- **2** Adds letter “a” to the end of list `first3letters`
- **3** Adds letter “c” to the end of list `first3letters`
- **4** Adds letter “b” at index 1 of list `first3letters`
- **5** Prints ['a', 'b', 'c']
- **6** Creates list with three elements in it
- **7** Adds elements “x”, “y”, “z” to the end of list `first3letters`
- **8** Prints ['a', 'b', 'c', 'x', 'y', 'z']
- **9** Extends list `last3letters` with the contents of the newly mutated list `first3letters`, by adding “a”, “b”, “c”, “x”, “y”, “z” to the `last3letters` list
- **10** Prints ['x', 'y', 'z', 'a', 'b', 'c', 'x', 'y', 'z']

With lists being mutable objects, certain actions you do on a list lead to the list object being changed. In [listing 25.2](#), the list `first3letters` is mutated every time you append to, insert to, and extend it. Similarly, the list `last3letters` is mutated when you extend it by the `first3letters` list.

#### Quick check 25.4

What's printed after each code snippet is executed?

**1**

```

one = [1]
one.append("1")
print(one)

```

**2**

```

zero = []
zero.append(0)
zero.append(["zero"])
print(zero)

```

**3**

```

two = []
three = []
three.extend(two)
print(three)

```

**4**

```

four = [1,2,3,4]
four.insert(len(four), 5)
print(four)
four.insert(0, 0)
print(four)

```

#### 25.5. REMOVING ITEMS FROM A LIST: POP

Lists aren't useful if you can only add items to them. They'll keep growing and quickly become unmanageable. Having mutable objects when removing items is also useful so you don't make copies with every change. For example, as you keep your grocery list, you want to remove items from it as you purchase them so you know that you don't need to look for them anymore. Each time you remove an item, you can keep the same list instead of transcribing all items you still need to a new list.

You can remove items from a list with the operation `pop()`, as shown in [list-](#)

senting the index whose value you want to remove with `L.pop(i)`. When removed, the list that the element is removed from is mutated. All elements after the one removed are shifted by one spot to replace the one removed. This operation is a function and returns the element removed.

Listing 25.3. Removing from a list

```
polite = ["please", "and", "thank", "you"]
print(polite.pop())
print(polite)
print(polite.pop(1))
print(polite)
```

- 1 List of four elements
- 2 Prints “you” because `pop()` returns the value of the element at the last index
- 3 Prints [‘please’, ‘and’, ‘thank’] because `pop()` removed the last element
- 4 Prints “and” because `pop(1)` returns the value of the element at index 1
- 5 Prints [‘please’, ‘thank’] because the preceding line removed the element at index 1 (second element in the list)

As with adding items to a list, removing items from the list also mutates the list. Every operation that removes an item changes the list on which the operation is performed. In listing 25.3, every time you print the list, you print the mutated list.

Quick check 25.5

Q1:

What’s printed when this code snippet is executed?

```
pi = [3, ".", 1, 4, 1, 5, 9]
pi.pop(1)
print(pi)
pi.pop()
print(pi)
pi.pop()
print(pi)
```

25.6. CHANGING AN ELEMENT VALUE

So far, you can add and remove items in a list. With mutability, you can even modify existing object elements in the list to change their values. For example, in your grocery list, you realize that you need cheddar instead of mozzarella cheese. As before, because of the mutability property, instead of copying the list over to another paper with the one item changed, it makes more sense to modify the list you currently have by replacing *mozzarella cheese* with *cheddar cheese*.

To modify an element in a list, you first need to access the element itself and then assign it a new value. You access an element by using its index. For example `L[0]` refers to the first element. The square brackets you used when you created a list now have this new purpose when placed to the right of the list variable name. The following listing shows how to do this.

Listing 25.4. Modifying an element value

```
colors = ["red", "blue", "yellow"]
colors[0] = "orange"
print(colors)
colors[1] = "green"
print(colors)
colors[2] = "purple"
print(colors)
```

- 1 Initializes list of strings
- 2 Changes “red” to “orange”
- 3 Prints [‘orange’, ‘blue’, ‘yellow’] because line 2 changed the element at index 0
- 4 Changes “blue” to “green” on the mutated list with element at index 0 now “orange”
- 5 Prints [‘orange’, ‘green’, ‘yellow’] because line 4 changed the element at index 1
- 6 Changes “yellow” to “purple” on mutated list with element

- 7 Prints ['orange', 'green', 'purple'] because line 6 changed the element at index 2

Elements in mutable lists can be modified to contain different values. After a list is mutated, every operation you do from then on is done on the mutated list.

#### Quick check 25.6

If `L` is a list of integers that initially contains the numbers shown in the first line of code that follows, what's the value of `L` after each of the four subsequent operations are done in order?

```
L = [1, 2, 3, 5, 7, 11, 13, 17]
```

1

```
L[3] = 4
```

2

```
L[4] = 6
```

3

```
L[-1] = L[0] (Recall how negative indices work from lesson 7.)
```

4

```
L[0] = L[1] + 1
```

#### SUMMARY

In this lesson, my objective was to teach you a new data type, a Python list. A list is a mutable object whose value can change. A list contains elements, and you can add to it, remove from it, change element values, and perform operations on the entire list. Here are the major takeaways:

- Lists can be empty or can contain elements.
- You can add an element to the end of a list, at a specific index, or you can extend it by more than one element.
- You can remove elements from the list, from the end or from a specific index.
- You can change element values.
- Every action mutates the list, so the list object changes without you having to reassign it to another variable.

Let's see if you got this...

#### Q25.1

You start out with the following empty list, intending to contain the items in a restaurant menu:

```
menu = []
```

1. Write one or more commands that mutate the list, so that the list menu contains

```
["pizza", "beer", "fries", "wings", "salad"].
```

2. Continuing on, write one or more commands that mutate the list to contain

```
["salad", "fries", "wings", "pizza"].
```

3. Finally, write one or more commands that mutate the list, so that it contains

```
["salad", "quinoa", "steak"].
```


#### Q25.2

Write a function named `unique`. It takes in one parameter, a list named `L`. The function doesn't mutate `L` and returns a new list containing only the unique elements in `L`.

Write a function named `common`. It takes in two parameters, lists named `L1` and `L2`. The function doesn't mutate `L1` or `L2`. It returns `True` if every unique element in `L1` is in `L2` and if every unique element in `L2` is in `L1`. It returns `False` otherwise. Hint: try to reuse your function from Q25.2. For example,

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

[Recommended](#) / [Playlists](#) / [History](#) / [Topics](#) / [Settings](#) / [Get the App](#) / [Sign Out](#)

 [PREV](#)  
[Lesson 24. Mutable and immutable objects](#)

[Lesson 26. Advanced operations with lists](#) 