Note

In the Notes, any code will be in blue, input is green and any output in red.

A list is a sequence.

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called *elements* or sometimes *items*.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ("[" and "]"):

```
[10, 20, 30, 40]
```

```
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings.

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is *nested*.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
empty = []
print(cheeses, numbers, empty)
```

```
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Lists are mutable.

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
print(cheeses[0])
```

```
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
numbers = [17, 123]
```

```
numbers[1] = 5

print(numbers)

[17, 5]
```

The one-th element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a *mapping*; each index "maps to" one of the elements.

List indices work the same way as string indices:

• Any integer expression can be used as an index.

• If you try to read or write an element that does not exist, you get an **IndexError**.

• If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists.

```
cheeses = ['Cheddar', 'Edam', 'Gouda']

'Edam' in cheeses

True

'Brie' in cheeses

False
```

List methods

Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

```
t = ['a', 'b', 'c']

t.append('d')

print(t)

['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

```
t1 = ['a', 'b', 'c']

t2 = ['d', 'e']

t1.extend(t2)

print(t1)

['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
t = ['d', 'c', 'e', 'b', 'a']

t.sort()

print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None.

If you accidentally write t = t.sort(), you will be disappointed with the result.

Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop**:

```
t = ['a', 'b', 'c']

x = t.pop(1)

print(t)
```

```
['a', 'c']
```

```
print(x)
```

```
b
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the **del** operator:

```
t = ['a', 'b', 'c']

del t[1]

print(t)
```

```
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use remove:

```
t = ['a', 'b', 'c']

t.remove('b')

print(t)
```

```
['a', 'c']
```

The return value from remove is None.