Method	Description
`startswith()`	If the string starts with the specified value, return `True`
`strip()`	Return a trimmed version of the string

1.3.3 Iterable Objects and Dictionaries

In Python, an **iterable object** is one that contains a collection of **elements** and defines, for each element, which element is next. In the following sections, we will consider some built-in iterable classes (types).

Box 1.1 Further Reading

- Python Community (2024a; § The Python Tutorial: 9 Classes), on classes, objects, and methods
- Python Community (2024a; § Python Standard Library: Built-in Types), on the basic built-in types

1.4 Lists

The **list** class defines an ordered set of elements. These elements can be of any class, and do not need to match within a list. Lists can be nested to create a list of lists. The basic syntax for creating a list of elements ex is `[e1, e2, ..., en]`. Consider the following list assignments:

```
int_list = [3, 9, 3, -4, 0]          # Duplication allowed
str_list = ["foo", "bar", "baz"]
com_list = [int_list, str_list]      # List of lists
mix_list = [8.41, "foo", [7]]        # Mixing element types
```

1.4.1 Accessing List Elements

Because the elements of a list have an order, they can be referred to via an **index**, a mapping of integers to elements. In Python, the first element in the list has index `0` and subsequent elements have indices of increasing values, `1`, `2`, `3`, and so on. The syntax for accessing the element with index `i` of a list `l` is `l[i]`. For instance, elements from the previously defined lists can be accessed as follows:

```
int_list[0]                          # => 3
int_list[3]                          # => -4
str_list[2]                          # => "baz"
mix_list[2]                          # => [7]
```

Negative indices are used to access elements from the end of a list. For instance, for `int_list` above,

```
int_list[-1]                    # => 0
int_list[-2]                    # => -4
```

This is particularly useful when we want to access the last element of a list, which we see has index `-1`.

A selection of elements from a list can be accessed via **slicing**, which has the syntax `l[start:stop]` or `l[start:stop:step]`. For instance,

```
l = [0, 1, 2, 3, 4]
l[0:3]                          # => [0, 1, 2]
l[2:4]                          # => [2, 3]
l[0:-1]                         # => [0, 1, 2, 3] (no last item!)
l[0:]                           # => [0, 1, 2, 3, 4]
l[0::2]                         # => [0, 2, 4] (every two elements)
```

It is important to note that the slice does not include the `stop` index; rather, the slice's last value is from index `stop-1`. As we see in the third slice example, this means the normal syntax for slicing through the final element (i.e., the element with index `-1`) does not include that element. To include the final element, leave off an index for `stop`, as shown in the fourth and fifth examples.

1.4.2 Mutability

Lists are **mutable**; that is, they can be mutated (changed). This is unlike most built-in types, which are **immutable** and cannot be changed. The mutability for frequently used built-in types is shown in table 1.5.

Table 1.5. Mutability of commonly used built-in types.

Data Type	Built-in Class	Mutability
Numbers	`int`, `float`, `complex`	Immutable
Strings	`str`	Immutable
Tuples	`tuple`	Immutable
Booleans	`bool`	Immutable
Lists	`list`	Mutable
Dictionaries	`dict`	Mutable
Sets	`set`	Mutable

The mutability of lists allows us to change their elements. The syntax for assigning a new value `v` to an element with index `i` of a list `l` is `l[i] = v`. For instance,

```
l = ["Hello", "World", "!"]
l[1] = "Stranger"
print(l)
```

returns

```
['Hello', 'Stranger', '!']
```

Note that although strings are immutable, a list of strings is mutable. This means `"Stranger"` is not at the same location in memory as was `"World"`.

1.4.3 Methods

Lists have several methods for mutating themselves, which are given in table 1.6.

Table 1.6. Commonly used list methods for a list `l`.

Method	Description
`l.append(item)`	Append `item` to the end of `l`
`l.clear()`	Remove all items from `l`
`l.extend(iterable)`	Concatenate `l` with the contents of `iterable`
`l.index(x[, start[, end]])`	Return the index of the first instance of `x` in `l[start:end]`
`l.insert(index, item)`	Insert `item` into `l` at `index`
`l.pop(index)`	Return and remove the item at `index`
`l.pop()`	Return and remove the last item
`l.remove(item)`	Remove `item`'s first occurrence
`l.reverse()`	Reverse the items of `l`
`l.sort(key=None, reverse=False)`	Sort the items of `l`

For example, an element can be inserted into a list as follows:

```
l = ["zero", "one", "three"]
l.insert(2, "two")
print(l)
```

which returns

```
['zero', 'one', 'two', 'three']
```

When using most list methods, we often do not assign the returned value from the expression. This is because most of these expressions return a value of `None`. For instance, from the previous example,

```
print(l.insert(2, "two"))
```

returns

```
None
```

Such methods are simply operating on the original list object and do not return that object. This is a common idiom in Python programming, and many mutable classes behave similarly.

Example 1.3

Write a program that removes the second occurrence of the element `3` from the following list:

```
l = [1, 2, 3, 0, 3, 4, 3]
```

The `remove()` method might seem promising, but it only removes the first occurrence of the element. Instead, let's identify the index of the second occurrence. The `index(x[, start[, end]])` method allows us to identify the index of the first occurrence or the first occurrence between `start` and `end`. So our strategy is to find the index `i_first` of the first occurrence with `index()`, then narrow our search to the rest of the list after `i_first` to the end of the list, identifying the second index `i_second`. Finally, we can remove the element at `i_second` with the pop method.

The following program implements this strategy.

```
l = [1, 2, 3, 0, 3, 4, 3]
x = 3                              # element we are removing
i_first = l.index(x)               # first occurrence index
i_second = l.index(x, i_first+1)   # second occurrence index
l.pop(i_second)                    # removes second occurrence
print(f"l without second {x}: {l}")
```

This prints

```
l without second 3: [1, 2, 3, 0, 4, 3]
```

1.5 Tuples and Ranges

Python has a built-in **tuple** class `tuple` is very similar to a `list` in that it is an ordered collection of elements. The term "tuple" is a generalization of the terms "single," "double," "triple," "quadruple," and so on. The primary difference between a tuple and a list is that a tuple is immutable, so its elements can't be changed. The syntax for a tuple literal of elements e*x* is `(e1, e2, ..., en)`. The elements can each be of any type, including tuples. For example, the following statements return tuples:

```
(0, 1, 2, 4, 5)
("foo", "bar", "baz")
([0, 1], [2, 3])
((0, 1), (2, 3))
(0, "foo", [1, 2], (3, 4))
```

Elements of a tuple can be accessed via the same syntax as is used for lists, including slicing. For instance,

```
t = (0, 1, 2)
t[1]            # => 1
t[0:2]          # => (0, 1)
t[1:]           # => (1, 2)
```