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]
```

1.7 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)
```

Because tuples are immutable, there are only two built-in tuple methods, `count()` and `index()`. The `count()` method returns the number of times its argument occurs in the tuple. For instance,

```
t = (-7, 0, 7, -7, 0, 0)
t.count(-7)     # => 2
```

The `index()` method returns the index of the first occurrence of its argument. For instance,

```
t = ("foo", "bar", "baz", "foo", "bar", "baz", "baz")
t.index("baz")  # => 2
```

The **range** built-in type is a compact way or representing sequences of integers. A `range` can be constructed with the `range(start, stop, step)` constructor function, as in the following examples:

```
list(range(0, 3, 1))    # => [0, 1, 2]
list(range(2, 6, 1))    # => [2, 3, 4, 5]
list(range(0, 3))       # => [0, 1, 2] (step=1 by default)
list(range(3))          # => [0, 1, 2] (start=0 by default)
```

Note that we have wrapped the ranges in `list()` functions, which converted each range to a list. This was only so we can see the values it represents; alone, an expression like `range(0, 3)` returns itself. This is why a range is such a compact data point—all that needs to be stored in memory are the `start`, `stop`, and `step` arguments because the intermediate values are implicit.

1.8 Dictionaries

The built-in Python **dictionary** class `dict` is an unordered collection of elements, each of which has a unique **key** and a **value**. A key can be any immutable object, but a string is most common. A value can be any object. The basic syntax to create a `dict` object with keys `k`*x* and values `v`*x* is `{k1: v1, k2: v2, ...}`. For instance, we can define a `dict` as follows:

```
d = {"foo": 5, "bar": 1, "baz": -3}
```

Accessing a value requires its key. To access a value in dictionary `d` with key `k`, use the syntax `d[k]`. For example,