In the next section, you'll learn about a third sequence type with one very big difference from strings and tuples: mutability.

### Review Exercises

*You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.*

1. Create a tuple literal named `cardinal_numbers` that holds the strings `"first"`, `"second"` and `"third"`, in that order.
2. Using index notation and `print()`, display the string at index `1` in `cardinal_numbers`.
3. Unpack the values in `cardinal_numbers` into three new strings named `position1`, `position2` and `position3` in a single line of code, then print each value on a separate line.
4. Create a tuple called `my_name` that contains the letters of your name by using `tuple()` and a string literal.
5. Check whether or not the character `"x"` is in `my_name` using the `in` keyword.
6. Create a new tuple containing all but the first letter in `my_name` using slicing notation.

Leave feedback on this section »

## 9.2 Lists Are Mutable Sequences

The `list` data structure is another sequence type in Python. Just like strings and tuples, lists contain items that are indexed by integers, starting with `0`.

On the surface, lists look and behave a lot like tuples. You can use index and slicing notation with lists, check for the existence of an element using `in`, and iterate over lists with a `for` loop.

Unlike tuples, however, lists are **mutable**, meaning you can change

the value at an index even after the list has been created.

In this section, you will learn how to create lists and compare them with tuples.

## Creating Lists

A **list literal** looks almost exactly like a tuple literal, except that it is surrounded with square brackets (`[` and `]`) instead of parentheses:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> type(colors)
<class 'list'>
```

When you inspect a list, Python displays it as a list literal:

```
>>> colors
['red', 'yellow', 'green', 'blue']
```

Like tuples, lists values are not required to be of the same type. The list literal `["one", 2, 3.0]` is perfectly valid.

Aside from list literals, you can also use the `list()` built-in to create a new list object from any other sequence. For instance, the tuple `(1, 2, 3)` can be passed to `list()` to create the list `[1, 2, 3]`:

```
>>> list((1, 2, 3))
[1, 2, 3]
```

You can even create a list from a string:

```
>>> list("Python")
['P', 'y', 't', 'h', 'o', 'n']
```

Each letter in the string becomes an element of the list.

There is more useful way to create a list from a string. You can create a list from a string of a comma-separated list of items using the string object's `.split()` method:

```
>>> groceries = "eggs, milk, cheese"
>>> grocery_list = groceries.split(", ")
>>> grocery_list
['eggs', 'milk', 'cheese']
```

The string argument passed to `.split()` is called the separator. By changing the separator you can split strings into lists in numerous ways:

```
>>> # Split string on semi-colons
>>> "a;b;c".split(";")
['a', 'b', 'c']

>>> # Split string on spaces
>>> "The quick brown fox".split(" ")
['The', 'quick', 'brown', 'fox']

>>> # Split string on multiple characters
>>> "abbaabba".split("ba")
['ab', 'ab', '']
```

In the last example above, the string is split around occurrences of the substring `"ba"`, which occurs first at index `2` and again at index `6`. The separator has two characters, only the characters at indices `1`, `2`, `5`, and `6` become elements of the list.

`.split()` always returns a string whose length is one more than the number of separators contained in the string. The string `"abbaabba"` contains two instances of the separator `"ba"` so the list returned by `split()` has three elements. Since the third separator isn't followed by any other characters, the third element of the list is set to the empty string.

If the separator is not contained in the string at all, `.split()` returns a list with the string as its only element:

```
>>> "abbaabba".split("c")
['abbaabba']
```

In all, you've seen three ways to create a list:

1. A list literal
2. The `list()` built-in
3. The string `.split()` method

Lists support the all of the same operations supported by tuples.

## Basic List Operations

Indexing and slicing operations work on lists the same way they do on tuples.

You can access list elements using index notation:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1]
2
```

You can create a new list from an existing once using slice notation:

```
>>> numbers[1:3]
[2, 3]
```

You can check for the existence of list elements using the `in` operator:

```
>>> # Check existence of an element
>>> "Bob" in numbers
False
```

Because lists are iterable, you can iterate over them with a `for` loop.

```
>>> # Print only the even numbers in the list
>>> for number in numbers:
...     if number % 2 == 0:
```

```
...         print(number)
...
2
4
```

The major difference between lists and tuples is that elements of lists may be changed, but elements of tuples can not.

## Changing Elements in a List

Think of a list as a sequence of numbered slots. Each slot holds a value, and every slot must be filled at all times, but you can swap out the value in a given slot with a new one whenever you want.

The ability to swap values in a list for other values is called **mutability**. Lists are **mutable**. The elements of tuples may not be swapped for new values, so tuples are said to be **immutable**.

To swap a value in a list with another, assign the new value to a slot using index notation:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[0] = "burgundy"
```

The value at index `0` changes from `"red"` to `"burgundy"`:

```
>>> colors
['burgundy', 'yellow', 'green', 'blue']
```

You can change several values in a list at once with a **slice assignment**:

```
>>> colors[1:3] = ["orange", "magenta"]
>>> colors
['burgundy', 'orange', 'magenta', 'blue']
```

`colors[1:3]` selects the slots with indices `1` and `2`. The values in these slots are assigned to `"orange"` and `"magenta"`, respectively.

The list assigned to a slice does not need to have the same length as the slice. For instance, you can assign a list of three elements to a slice with two elements:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[1:3] = ["orange", "magenta", "aqua"]
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
```

The values `"orange"` and `"magenta"` replace the original values `"yellow"` and `"green"` in `colors` at the indices `1` and `2`. Then a new slot is created at index `4` and `"blue"` is assigned to this index. Finally, `"aqua"` is assigned to index `3`.

When the length of the list being assigned to the slice is less than the length of the slice, the overall length of the original list is reduced:

```
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
>>> colors[1:4] = ["yellow", "green"]
>>> colors
['red', 'yellow', 'green', 'blue']
```

The values `"yellow"` and `"green"` replace the values `"orange"` and `"magenta"` in `colors` at the indices `1` and `2`. Then the value at index `3` is replaced with the value `"blue"`. Finally, the slot at index `4` is removed from `colors` entirely.

The above examples show how to change, or **mutate**, lists using index and slice notation. There are also several list methods that you can use to mutate a list.

## List Methods For Adding and Removing Elements

Although you can add and remove elements with slice notation, list methods provide a more natural and readable way to mutate a list.

We'll look at several list methods, starting with how to insert a single

value into a list at a specified index.

#### `list.insert()`

The `list.insert()` method is used to insert a single new value into a list. It takes two parameters, an index `i` and a value `x`, and inserts the value `x` at index `i` in the list.

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> # Insert "orange" into the second position
>>> colors.insert(1, "orange")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
```

There are a couple of important observations to make about this example.

The first observation applies to all list methods. To use them, you first write the name of the list you want to manipulate, followed by a dot (`.`) and then the name of the list method.

So, to use `insert()` on the `colors` list, you must write `colors.insert()`. This works just like string and number methods do.

Next, notice that when the value `"orange"` is inserted at the index `1`, the value `"yellow"` and all following values are shifted to the right.

If the value for the index parameter of `.insert()` is larger than the greatest index in the list, the value is inserted at the end of the list:

```
>>> colors.insert(10, "violet")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'violet']
```

Here the value `"violet"` is actually inserted at index `5`, even though `.insert()` was called with `10` for the index.

You can also use negative indices with `.insert()`:

```
>>> colors.insert(-1, "indigo")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This inserts `"indigo"` into the slot at index `-1` which is the last element of the list. The value `"violet"` is shifted to the right by one slot.

**Important**

When you `.insert()` an item into a list, you do not need to assign the result to the original list.

For example, the following code actually erases the colors list:

```
>>> colors = colors.insert(-1, "indigo")
>>> print(colors)
None
```

`.insert()` is said to alter `colors` **in place**. This is true for all list methods that do not return a value.

If you can insert a value at a specified index, it only makes sense that you can also remove an element at a specified index.

#### `list.pop()`

The `list.pop()` method takes one parameter, an index `i`, and removes the value from the list at that index. The value that is removed is returned by the method:

```
>>> color = colors.pop(3)
>>> color
'green'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Here, the value `"green"` at index `3` is removed and and assigned to the variable `color`. When you inspect the `colors` list, you can see that the string `"green"` has indeed been removed.

Unlike `.insert()`, Python raises an `IndexError` if you pass to `.pop()` an argument larger than the last index:

```
>>> colors.pop(10)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    colors.pop(10)
IndexError: pop index out of range
```

Negative indices also work with `.pop()`:

```
>>> colors.pop(-1)
'violet'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

If you do not pass a value to `.pop()`, it removes the last item in the list:

```
>>> colors.pop()
'indigo'
>>> colors
['red', 'orange', 'yellow', 'blue']
```

This way of removing the final element, by calling `.pop()` with no specified index, is generally considered the most Pythonic.

#### `list.append()`

The `list.append()` method is used to append an new element to the end of a list:

```
>>> colors.append("indigo")
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

After calling `.append()`, the length of the list increases by one and the value `"indigo"` is inserted into the final slot. Note that `.append()` alters the list in place, just like `.insert()`.

`.append()` is equivalent to inserting an element at an index greater than or equal to the length of the list. The above example could also have been written as follows:

```
>>> colors.insert(len(colors), "indigo")
```

`.append()` is both shorter and more descriptive than using `.insert()` this way, and is generally considered the more Pythonic way of added an element to the end of a list.

#### `list.extend()`

The `list.extend()` method is used to add several new elements to the end of a list:

```
>>> colors.extend(["violet", "ultraviolet"])
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet', 'ultraviolet']
```

`.extend()` takes a single parameter that must be an iterable type. The elements of the iterable are appended to the list in the same order that they appear in the argument passed to `.extend()`.

Just like `.insert()` and `.append()`, `.extend()` alters the list in place.

Typically, the argument passed to `.extend()` is another list, but it could also be a tuple. For example, the above example could be written as follows:

```
>>> colors.extend(("violet", "ultraviolet"))
```

The four list methods discussed in this section make up the most common methods used with lists. The following table serves to recap everything you have seen here:

| List Method | Description |
|---|---|
| `.insert(i, x)` | Insert the value `x` at index `i` |
| `.append(x)` | Insert the value `x` at the end of the list |

| List Method | Description |
|---|---|
| `.extend(iterable)` | Insert all the values of `iterable` at the end of the list, in order |
| `.pop(i)` | Remove and return the element at index `i` |

In addition to list methods, Python has a couple of useful built-in functions for working with lists of numbers.

## Lists of Numbers

One very common operation with lists of numbers is to add up all the values to get the total.

You can do this with a `for` loop:

```
>>> nums = [1, 2, 3, 4, 5]
>>> total = 0
>>> for number in nums:
...     total = total + number
...
>>> total
15
```

First you initialize the variable `total` to `0`, and then loop over each number is `nums` and add it to `total`, finally arriving at the value `15`.

Although this `for` loop is straightforward, there is a much more succinct way of doing this in Python:

```
>>> sum([1, 2, 3, 4, 5])
15
```

The built-in `sum()` function takes a list as an argument and returns the total of all the values in the list.

If the list passed to `sum()` contains any values that aren't numeric, a `TypeError` is raised:

```
>>> sum([1, 2, 3, "four", 5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Besides `sum()`, there are two other useful built-in functions for working with lists of numbers: `min()` and `max()`. These functions return the minimum and maximum values in the list, respectively:

```
>>> min([1, 2, 3, 4, 5 ])
1

>>> max([1, 2, 3, 4, 5])
5
```

Note that `sum()`, `min()`, and `max()` also work with tuples:

```
>>> sum((1, 2, 3, 4, 5))
15

>>> min((1, 2, 3, 4, 5))
1

>>> max((1, 2, 3, 4, 5))
5
```

The fact that `sum()`, `min()`, and `max()` are all built-in to Python tells you that they are used frequently. Chances are, you'll find yourself using them quite a bit in your own programs!

## List Comprehensions

Yet another way to create a list from an existing iterable is with a **list comprehension**:

```
>>> numbers = (1, 2, 3, 4, 5)
>>> squares = [num**2 for num in numbers]
```

```
>>> squares
[1, 4, 9, 16, 25]
```

A list comprehension is a short-hand for a `for` loop. In the example above, a tuple literal containing five numbers is created and assigned to the `numbers` variable. On the second line, a list comprehension loops over each number in `numbers`, squares each number, and adds it to a new list called `squares`.

To create the `sqaures` list using a traditional `for` loop involves first creating an empty list, looping over the numbers in `numbers`, and appending the square of each number to the list:

```
>>> squares = []
>>> for num in numbers:
...     sqaures.append(num**2)
...
>>> squares
[1, 4, 9, 16, 25]
```

List comprehensions are commonly used to convert values in one list to a different type.

For instance, suppose you needed to convert a list of strings containing floating point values to a list of `float` objects. The following list comprehensions achieves this:

```
>>> str_numbers = ["1.5", "2.3", "5.25"]
>>> float_numbers = [float(value) for value in str_numbers]
>>> float_numbers
[1.5, 2.3, 5.25]
```

List comprehensions are not unique to Python, but they are one of its many beloved features. If you find yourself creating an empty list, looping over some other iterable, and appending new items to the list, then chances are you can replace your code with a list comprehension!