# Chapter 7. Tuples and Lists

*The human animal differs from the lesser primates in his passion for lists.*

—H. Allen Smith

In the previous chapters, we started with some of Python's basic data types: booleans, integers, floats, and strings. If you think of those as atoms, the data structures in this chapter are like molecules. That is, we combine those basic types in more complex ways. You will use these every day. Much of programming consists of chopping and gluing data into specific forms, and these are your hacksaws and glue guns.

Most computer languages can represent a sequence of items indexed by their integer position: first, second, and so on down to the last. You've already seen Python *strings*, which are sequences of characters.

Python has two other sequence structures: *tuples* and *lists*. These contain zero or more elements. Unlike strings, the elements can be of different types. In fact, each element can be *any* Python object. This lets you create structures as deep and complex as you like.

Why does Python contain both lists and tuples? Tuples are *immutable*; when you assign elements (only once) to a tuple, they're baked in the cake and can't be changed. Lists are *mutable*, meaning you can insert and delete elements with great enthusiasm. I'll show many examples of each, with an emphasis on lists.

## Tuples

Let's get one thing out of the way first. You may hear two different pronunciations for *tuple*. Which is right? If you guess wrong, do you risk being considered a Python poseur? No worries. Guido van Rossum, the creator of Python, said via Twitter:

*I pronounce tuple too-pull on Mon/Wed/Fri and tub-pull on Tue/Thu/Sat. On Sunday I don't talk about them. :)*

## Create with Commas and ()

The syntax to make tuples is a little inconsistent, as the following examples demonstrate. Let's begin by making an empty tuple using `()`:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

To make a tuple with one or more elements, follow each element with a comma. This works for one-element tuples:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

You could enclose them in parentheses and still get the same tuple:

```
>>> one_marx = ('Groucho',)
>>> one_marx
('Groucho',)
```

Here's a little gotcha: if you have a single thing in parentheses and omit that comma, you would not get a tuple, but just the thing (in this example, the string `'Groucho'`):

```
>>> one_marx = ('Groucho')
>>> one_marx
'Groucho'
>>> type(one_marx)
<class 'str'>
```

If you have more than one element, follow all but the last one with a comma:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Python includes parentheses when echoing a tuple. You often don’t need them when you define a tuple, but using parentheses is a little safer, and it helps to make the tuple more visible:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

You do need the parentheses for cases in which commas might also have another use. In this example, you can create and assign a single-element tuple with just a trailing comma, but you can’t pass it as an argument to a function:

```
>>> one_marx = 'Groucho',
>>> type(one_marx)
<class 'tuple'>
>>> type('Groucho',)
<class 'str'>
>>> type(('Groucho',))
<class 'tuple'>
```

Tuples let you assign multiple variables at once:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

This is sometimes called *tuple unpacking*.

You can use tuples to exchange values in one statement without using a temporary variable:

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
>>> icecream
'swordfish'
>>>
```

## Create with tuple()

The `tuple()` conversion function makes tuples from other things:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

## Combine Tuples by Using `+`

This is similar to combining strings:

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

## Duplicate Items with *

This is like repeated use of `+`:

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

## Compare Tuples

This works much like list comparisons:

```
>>> a = (7, 2)
>>> b = (7, 2, 9)
>>> a == b
```

```
False
>>> a <= b
True
>>> a < b
True
```

## Iterate with for and in

Tuple iteration is like iteration of other types:

```
>>> words = ('fresh','out', 'of', 'ideas')
>>> for word in words:
...     print(word)
...
fresh
out
of
ideas
```

## Modify a Tuple

You can't! Like strings, tuples are immutable, so you can't change an existing one. As you saw just before, you can *concatenate* (combine) tuples to make a new one, as you can with strings:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 + t2
('Fee', 'Fie', 'Foe', 'Flop')
```

This means that you can appear to modify a tuple like this:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 += t2
>>> t1
('Fee', 'Fie', 'Foe', 'Flop')
```

But it isn't the same `t1`. Python made a new tuple from the original tuples pointed to by `t1` and `t2`, and assigned the name `t1` to this new tuple. You can see with `id()` when a variable name is pointing to a new value:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> id(t1)
4365405712
>>> t1 += t2
>>> id(t1)
4364770744
```

## Lists

Lists are good for keeping track of things by their order, especially when the order and contents might change. Unlike strings, lists are mutable. You can change a list in place, add new elements, and delete or replace existing elements. The same value can occur more than once in a list.

### Create with []

A list is made from zero or more elements, separated by commas and surrounded by square brackets:

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
>>> leap_years = [2000, 2004, 2008]
>>> randomness = ['Punxsatawney", {"groundhog": "Phil"}, "Feb. 2"}
```

The `first_names` list shows that values do not need to be unique.

---

**NOTE**

If you want to keep track of only unique values and don't care about order, a Python *set* might be a better choice than a list. In the previous example, `big_birds` could have been a set. We explore sets in Chapter 8.

---

## Create or Convert with list()

You can also make an empty list with the `list()` function:

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```

Python’s `list()` function also converts other *iterable* data types (such as tuples, strings, sets, and dictionaries) to lists. The following example converts a string to a list of one-character strings:

```
>>> list('cat')
['c', 'a', 't']
```

This example converts a tuple to a list:

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

## Create from a String with split()

As I mentioned earlier in “Split with split()”, use `split()` to chop a string into a list by some separator:

```
>>> talk_like_a_pirate_day = '9/19/2019'
>>> talk_like_a_pirate_day.split('/')
['9', '19', '2019']
```

What if you have more than one separator string in a row in your original string? Well, you get an empty string as a list item:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

If you had used the two-character separator string `//`, instead, you would get this:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

## Get an Item by [ *offset* ]

As with strings, you can extract a single value from a list by specifying its offset:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Again, as with strings, negative indexes count backward from the end:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

NOTE

The offset has to be a valid one for this list—a position you have assigned a value previously. If you specify an offset before the beginning or after the end, you'll get an exception (error). Here's what happens if we try to get the sixth Marx brother (offset `5` counting from `0`), or the fifth before the end:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> marxes[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

## Get Items with a Slice

You can extract a subsequence of a list by using a *slice*:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

A slice of a list is also a list.

As with strings, slices can step by values other than one. The next example starts at the beginning and goes right by 2:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Here, we start at the end and go left by 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

And finally, the trick to reverse a list:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

None of these slices changed the `marxes` list itself, because we didn't assign them to `marxes`. To reverse a list in place, use `list.reverse()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.reverse()
>>> marxes
['Harpo', 'Chico', 'Groucho']
```

---

**NOTE**

The `reverse()` function changes the list but doesn't return its value.

---

As you saw with strings, a slice can specify an invalid index, but will not cause an exception. It will "snap" to the closest valid index or return nothing:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[4:]
[]
>>> marxes[-6:]
['Groucho', 'Chico', 'Harpo']
>>> marxes[-6:-2]
['Groucho']
>>> marxes[-6:-4]
[]
```

## Add an Item to the End with append()

The traditional way of adding items to a list is to `append()` them one by one to the end. In the previous examples, we forgot Zeppo, but that's alright because the list is mutable, so we can add him now:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

## Add an Item by Offset with insert()

The `append()` function adds items only to the end of the list. When you want to add an item before any offset in the list, use `insert()`. Offset `0` inserts at the beginning. An offset beyond the end of the list inserts at the end, like `append()`, so you don't need to worry about Python throwing an exception:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.insert(2, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Harpo']
>>> marxes.insert(10, 'Zeppo')
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Harpo', 'Zeppo']
```

## Duplicate All Items with *

In Chapter 5, you saw that you can duplicate a string's characters with `*`. The same works for a list:

```
>>> ["blah"] * 3
['blah', 'blah', 'blah']
```

## Combine Lists by Using extend() or `+`

You can merge one list into another by using `extend()`. Suppose that a well-meaning person gave us a new list of Marxes called `others`, and we'd like to merge them into the main `marxes` list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
```

```
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Alternatively, you can use `+` or `+=`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

If we had used `append()`, `others` would have been added as a *single* list item rather than merging its items:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

This again demonstrates that a list can contain elements of different types. In this case, four strings, and a list of two strings.

## Change an Item by [ *offset* ]

Just as you can get the value of a list item by its offset, you can change it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

Again, the list offset needs to be a valid one for this list.

You can't change a character in a string in this way, because strings are immutable. Lists are mutable. You can change how many items a list contains as well as the items themselves.

## Change Items with a Slice

The previous section showed how to get a sublist with a slice. You can also assign values to a sublist with a slice:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
```

The righthand thing that you're assigning to the list doesn't even need to have the same number of elements as the slice on the left:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [7, 8, 9]
>>> numbers
[1, 7, 8, 9, 4]
```

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

Actually, the righthand thing doesn't even need to be a list. Any Python *it-erable* will do, separating its items and assigning them to list elements:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]
```

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'wat?'
>>> numbers
[1, 'w', 'a', 't', '?', 4]
```

## Delete an Item by Offset with del

Our fact checkers have just informed us that Gummo was indeed one of the Marx Brothers, but Karl wasn't, and that whoever inserted him ear-lier was very rude. Let's fix that:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Karl']
>>> marxes[-1]
'Karl'
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo']
```

When you delete an item by its position in the list, the items that follow it move back to take the deleted item's space, and the list's length decreases by one. If we deleted `'Chico'` from the last version of the `marxes` list, we get this as a result:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo']
>>> del marxes[1]
>>> marxes
['Groucho', 'Harpo', 'Gummo']
```

---

**NOTE**

`del` is a Python *statement*, not a list method—you don't say `marxes[-1].del()`. It's sort of the reverse of assignment (`=`): it detaches a name from a Python object and can free up the object's memory if that name were the last reference to it.

---

## Delete an Item by Value with remove()

If you're not sure or don't care where the item is in the list, use `remove()` to delete it by value. Goodbye, Groucho:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.remove('Groucho')
>>> marxes
['Chico', 'Harpo']
```

If you had duplicate list items with the same value, `remove()` deletes only the first one it finds.

## Get an Item by Offset and Delete It with pop()

You can get an item from a list and delete it from the list at the same time by using `pop()`. If you call `pop()` with an offset, it will return the item at that offset; with no argument, it uses `-1`. So, `pop(0)` returns the head (start) of the list, and `pop()` or `pop(-1)` returns the tail (end), as shown here:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

NOTE

It's computing jargon time! Don't worry, these won't be on the final exam. If you use `append()` to add new items to the end and `pop()` to remove them from the same end, you've implemented a data structure known as a *LIFO* (last in, first out) queue. This is more commonly known as a *stack*. `pop(0)` would create a *FIFO* (first in, first out) queue. These are useful when you want to collect data as they arrive and work with either the oldest first (FIFO) or the newest first (LIFO).

## Delete All Items with clear()

Python 3.3 introduced a method to clear a list of all its elements:

```
>>> work_quotes = ['Working hard?', 'Quick question!', 'Number one priorities!']
>>> work_quotes
['Working hard?', 'Quick question!', 'Number one priorities!']
>>> work_quotes.clear()
>>> work_quotes
[]
```

## Find an Item's Offset by Value with index()

If you want to know the offset of an item in a list by its value, use `index()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

If the value is in the list more than once, only the offset of the first one is returned:

```
>>> simpsons = ['Lisa', 'Bart', 'Marge', 'Homer', 'Bart']
>>> simpsons.index('Bart')
1
```

## Test for a Value with in

The Pythonic way to check for the existence of a value in a list is using `in`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

The same value may be in more than one position in the list. As long as it's in there at least once, `in` will return `True`:

```
>>> words = ['a', 'deer', 'a' 'female', 'deer']
>>> 'deer' in words
True
```

NOTE

If you check for the existence of some value in a list often and don't care about the order of items, a Python *set* is a more appropriate way to store and look up unique values. We talk about sets in Chapter 8.

## Count Occurrences of a Value with count()

To count how many times a particular value occurs in a list, use `count()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0
```

```
>>> snl_skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> snl_skit.count('cheeseburger')
3
```

## Convert a List to a String with join()

"Combine by Using join()" discussed `join()` in greater detail, but here's another example of what you can do with it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

You might be thinking that this seems a little backward. `join()` is a string method, not a list method. You can't say `marxes.join(', ')`, even though it seems more intuitive. The argument to `join()` is a string or any iterable sequence of strings (including a list), and its output is a string. If `join()` were just a list method, you couldn't use it with other iterable objects such as tuples or strings. If you did want it to work with any iterable type, you'd need special code for each type to handle the actual joining. It might help to remember—`join()` *is the opposite of* `split()`, as shown here:

```
>>> friends = ['Harry', 'Hermione', 'Ron']
>>> separator = ' * '
>>> joined = separator.join(friends)
>>> joined
'Harry * Hermione * Ron'
>>> separated = joined.split(separator)
>>> separated
['Harry', 'Hermione', 'Ron']
>>> separated == friends
True
```

## Reorder Items with sort() or sorted()

You’ll often need to sort the items in a list by their values rather than their offsets. Python provides two functions:

- The list method `sort()` sorts the list itself, *in place*.
- The general function `sorted()` returns a sorted *copy* of the list.

If the items in the list are numeric, they’re sorted by default in ascending numeric order. If they’re strings, they’re sorted in alphabetical order:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Chico', 'Groucho', 'Harpo']
```

`sorted_marxes` is a new list, and creating it did not change the original list:

```
>>> marxes
['Groucho', 'Chico', 'Harpo']
```

But calling the list function `sort()` on the `marxes` list does change `marxes`:

```
>>> marxes.sort()
>>> marxes
['Chico', 'Groucho', 'Harpo']
```

If the elements of your list are all of the same type (such as strings in `marxes`), `sort()` will work correctly. You can sometimes even mix types—for example, integers and floats—because they are automatically converted to one another by Python in expressions:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

The default sort order is ascending, but you can add the argument `reverse=True` to set it to descending:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

## Get Length with len()

`len()` returns the number of items in a list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> len(marxes)
3
```

## Assign with =

When you assign one list to more than one variable, changing the list in one place also changes it in the other, as illustrated here:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

So what's in `b` now? Is it still `[1, 2, 3]`, or `['surprise', 2, 3]`? Let's see:

```
>>> b
['surprise', 2, 3]
```

Remember the box (object) and string with note (variable name) analogy in Chapter 2? `b` just refers to the same list object as `a` (both name strings lead to the same object box). Whether we change the list contents by using the name `a` or `b`, it's reflected in both:

```
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

## Copy with copy(), list(), or a Slice

You can *copy* the values of a list to an independent, fresh list by using any of these methods:

- The list `copy()` method
- The `list()` conversion function
- The list slice `[:]`

Our original list will be `a` again. We make `b` with the list `copy()` function, `c` with the `list()` conversion function, and `d` with a list slice:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

Again, `b`, `c`, and `d` are *copies* of `a`: they are new objects with their own values and no connection to the original list object `[1, 2, 3]` to which `a` refers. Changing `a` does *not* affect the copies `b`, `c`, and `d`:

```
>>> a[0] = 'integer lists are boring'
>>> a
['integer lists are boring', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

## Copy Everything with deepcopy()

The `copy()` function works well if the list values are all immutable. As you've seen before, mutable values (like lists, tuples, or dicts) are references. A change in the original or the copy would be reflected in both.

Let's use the previous example but make the last element in list `a` the list `[8, 9]` instead of the integer `3`:

```
>>> a = [1, 2, [8, 9]]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> c
[1, 2, [8, 9]]
>>> d
[1, 2, [8, 9]]
```

So far, so good. Now change an element in that sublist in `a`:

```
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 10]]
>>> c
[1, 2, [8, 10]]
>>> d
```

```
[1, 2, [8, 10]]
```

The value of `a[2]` is now a list, and its elements can be changed. All the list-copying methods we used were *shallow* (not a value judgment, just a depth one).

To fix this, we need to use the `deepcopy()` function:

```
>>> import copy
>>> a = [1, 2, [8, 9]]
>>> b = copy.deepcopy(a)
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 9]]
```

`deepcopy()` can handle deeply nested lists, dictionaries, and other objects.

You'll read more about `import` in Chapter 9.

## Compare Lists

You can directly compare lists with the comparison operators like `==`, `<`, and so on. The operators walk through both lists, comparing elements at the same offsets. If list `a` is shorter than list `b`, and all of its elements are equal, `a` is less than `b`:

```
>>> a = [7, 2]
>>> b = [7, 2, 9]
>>> a == b
False
>>> a <= b
True
>>> a < b
True
```

## Iterate with for and in

In Chapter 6, you saw how to iterate over a string with `for`, but it's much more common to iterate over lists:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     print(cheese)
...
brie
gjetost
havarti
```

As before, `break` ends the `for` loop and `continue` steps to the next iteration:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('g'):
...         print("I won't eat anything that starts with 'g'")
...         break
...     else:
...         print(cheese)
...
brie
I won't eat anything that starts with 'g'
```

You can still use the optional `else` if the `for` completed without a `break`:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('x'):
...         print("I won't eat anything that starts with 'x'")
...         break
...     else:
...         print(cheese)
... else:
...     print("Didn't find anything that started with 'x'")
...
brie
gjetost
```

```
havarti
Didn't find anything that started with 'x'
```

If the initial `for` never ran, control goes to the `else` also:

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
... else:  # no break means no cheese
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Because the `cheeses` list was empty in this example, `for cheese in cheeses` never completed a single loop and its `break` statement was never executed.

## Iterate Multiple Sequences with zip()

There's one more nice iteration trick: iterating over multiple sequences in parallel by using the `zip()` function:

```
>>> days = ['Monday', 'Tuesday', 'Wednesday']
>>> fruits = ['banana', 'orange', 'peach']
>>> drinks = ['coffee', 'tea', 'beer']
>>> desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']
>>> for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
...     print(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)
...
Monday : drink coffee - eat banana - enjoy tiramisu
Tuesday : drink tea - eat orange - enjoy ice cream
Wednesday : drink beer - eat peach - enjoy pie
```

`zip()` stops when the shortest sequence is done. One of the lists (`desserts`) was longer than the others, so no one gets any pudding unless we extend the other lists.

Chapter 8 shows you how the `dict()` function can create dictionaries from two-item sequences like tuples, lists, or strings. You can use `zip()` to walk through multiple sequences and make tuples from items at the same offsets. Let’s make two tuples of corresponding English and French words:

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> french = 'Lundi', 'Mardi', 'Mercredi'
```

Now, use `zip()` to pair these tuples. The value returned by `zip()` is itself not a tuple or list, but an iterable value that can be turned into one:

```
>>> list( zip(english, french) )
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

Feed the result of `zip()` directly to `dict()` and voilà: a tiny English-French dictionary!

```
>>> dict( zip(english, french) )
{'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

## Create a List with a Comprehension

You saw how to create a list with square brackets or the `list()` function. Here, we look at how to create a list with a *list comprehension*, which incorporates the `for` / `in` iteration that you just saw.

You could build a list of integers from `1` to `5`, one item at a time, like this:

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

Or, you could also use an iterator and the `range()` function:

```
>>> number_list = []
>>> for number in range(1, 6):
...     number_list.append(number)
...
>>> number_list
[1, 2, 3, 4, 5]
```

Or, you could just turn the output of `range()` into a list directly:

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

All of these approaches are valid Python code and will produce the same result. However, a more Pythonic (and often faster) way to build a list is by using a *list comprehension*. The simplest form of list comprehension looks like this:

```
[expression for item in iterable]
```

Here's how a list comprehension would build the integer list:

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

In the first line, you need the first `number` variable to produce values for the list: that is, to put a result of the loop into `number_list`. The second `number` is part of the `for` loop. To show that the first `number` is an expression, try this variant:

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

The list comprehension moves the loop inside the square brackets. This comprehension example really wasn't simpler than the previous example, but there's more that you can do. A list comprehension can include a conditional expression, looking something like this:

```
[expression for item
in iterable if condition]
```

Let's make a new comprehension that builds a list of only the odd numbers between `1` and `5` (remember that `number % 2` is `True` for odd numbers and `False` for even numbers):

```
>>> a_list = [number for number in range(1,6) if number % 2 == 1]
>>> a_list
[1, 3, 5]
```

Now, the comprehension is a little more compact than its traditional counterpart:

```
>>> a_list = []
>>> for number in range(1,6):
...     if number % 2 == 1:
...         a_list.append(number)
...
>>>  a_list
[1, 3, 5]
```

Finally, just as there can be nested loops, there can be more than one set of `for ...` clauses in the corresponding comprehension. To show this, let's first try a plain old nested loop and print the results:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...         print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

Now, let's use a comprehension and assign it to the variable `cells`, making it a list of `(row, col)` tuples:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

By the way, you can also use *tuple unpacking* to get the `row` and `col` values from each tuple as you iterate over the `cells` list:

```
>>> for row, col in cells:
...     print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

The `for row ...` and `for col ...` fragments in the list comprehension could also have had their own `if` tests.

## Lists of Lists

Lists can contain elements of different types, including other lists, as illustrated here:

```
>>> small_birds = ['hummingbird', 'finch']
>>> extinct_birds = ['dodo', 'passenger pigeon', 'Norwegian Blue']
>>> carol_birds = [3, 'French hens', 2, 'turtledoves']
>>> all_birds = [small_birds, extinct_birds, 'macaw', carol_birds]
```

So what does `all_birds` , a list of lists, look like?

```
>>> all_birds
[['hummingbird', 'finch'], ['dodo', 'passenger pigeon', 'Norwegian Blue'], 'macaw
[3, 'French hens', 2, 'turtledoves']]
```

Let's look at the first item in it:

```
>>> all_birds[0]
['hummingbird', 'finch']
```

The first item is a list: in fact, it's `small_birds` , the first item we specified when creating `all_birds` . You should be able to guess what the second item is:

```
>>> all_birds[1]
['dodo', 'passenger pigeon', 'Norwegian Blue']
```

It's the second item we specified, `extinct_birds` . If we want the first item of `extinct_birds` , we can extract it from `all_birds` by specifying two indexes:

```
>>> all_birds[1][0]
'dodo'
```

The `[1]` refers to the list that's the second item in `all_birds` , and the `[0]` refers to the first item in that inner list.

## Tuples Versus Lists

You can often use tuples in place of lists, but they have many fewer functions—there is no `append()` , `insert()` , and so on—because they can't be modified after creation. Why not just use lists instead of tuples everywhere?

- Tuples use less space.
- You can't clobber tuple items by mistake.
- You can use tuples as dictionary keys (see Chapter 8).

- *Named tuples* (see "Named Tuples") can be a simple alternative to objects.

I won't go into much more detail about tuples here. In everyday programming, you'll use lists and dictionaries more.

## There Are No Tuple Comprehensions

Mutable types (lists, dictionaries, and sets) have comprehensions. Immutable types like strings and tuples need to be created with the other methods listed in their sections.

You might have thought that changing the square brackets of a list comprehension to parentheses would create a tuple comprehension. And it would appear to work because there's no exception if you type this:

```
>>> number_thing = (number for number in range(1, 6))
```

The thing between the parentheses is something else entirely: a *generator comprehension*, and it returns a *generator object*:

```
>>> type(number_thing)
<class 'generator'>
```

I'll get into generators in more detail in "Generators". A generator is one way to provide data to an iterator.

## Coming Up

They're so swell, they get their own chapter: *dictionaries* and *sets*.

## Things to Do

Use lists and tuples with numbers (Chapter 3) and strings (Chapter 5) to represent elements in the real world with great variety.

7.1 Create a list called `years_list`, starting with the year of your birth, and each year thereafter until the year of your fifth birthday. For example, if you were born in 1980, the list would be `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`. If you're less than five years old and reading this book, I don't know what to tell you.

7.2 In which year in `years_list` was your third birthday? Remember, you were 0 years of age for your first year.

7.3 In which year in `years_list` were you the oldest?

7.4 Make a list called `things` with these three strings as elements: `"mozzarella"`, `"cinderella"`, `"salmonella"`.

7.5 Capitalize the element in `things` that refers to a person and then print the list. Did it change the element in the list?

7.6 Make the cheesy element of `things` all uppercase and then print the list.

7.7 Delete the disease element from `things`, collect your Nobel Prize, and print the list.

7.8 Create a list called `surprise` with the elements `"Groucho"`, `"Chico"`, and `"Harpo"`.

7.9 Lowercase the last element of the `surprise` list, reverse it, and then capitalize it.

7.10 Use a list comprehension to make a list called `even` of the even numbers in `range(10)`.

7.11 Let's create a jump rope rhyme maker. You'll print a series of two-line rhymes. Start with this program fragment:

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
        ("fog", "walk the dog"),
        ("fun", "say we're done"),
        ]
    start2 = "Someone better"
```

For each tuple (`first, second`) in `rhymes`:

For the first line:

- Print each string in `st`

`art1`, capitalized and followed by an exclamation point and a space.

- Print `first`, also capitalized and followed by an exclamation

mation point.

For the second line:

- Print `start2` and a space.
- Print `second` and a period.