General

One element tuples

One-element tuples look like:

```
1,
```

The essential part of the notation here is the trailing comma. As for any expression, parentheses are optional, so you may write one-element tuples like:

```
(1, )
```

But it is the comma, not the parentheses, that define the tuple. Take a look at the following example:

```
>>> sequence = (3)              # an integer
>>> reeks
3
>>> isinstance(reeks, tuple)
False
>>> isinstance(reeks, int)
True
>>> reeks = (3, )               # a tuple
>>> isinstance(reeks, tuple)
True
```

The comma is essential, because Python otherwise interprets the notation `(object)` as the object itself.

Repeating the elements of a tuple

Just as strings, tuples can also be multiplied with a positive integer. This creates a new tuple that contains a repetition of the elements in the original tuple.

```
>>> tuple = (2, 3)
>>> tuple * 4
(2, 3, 2, 3, 2, 3, 2, 3)
>>> element = (0, )
>>> 6 * element
(0, 0, 0, 0, 0, 0)
```

Grouping the elements of a list

Python offers multiple solutions for grouping the elements in a list into group of n elements. For example, you may use a list comprehension to solve this problem.

```
>>> some_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> [(some_list[i], some_list[i + 1]) for i in range(0, len(some_list), 2)]
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

You may also use the built-in function `zip` that simultaneously iterates two or more iterable objects. Each iteration step, the function returns a tuple containing the i-th elements of the iterable objects that are passed to it.

If you simultaneously iterate over the list containing all elements at even positions (some_list[::2]) and the list containing all elements at odd positions (some_list[1::2]), you obtain exactly the same result.

```
>>> some_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> zip(some_list[::2], some_list[1::2])
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

All en any

The Python functions `any` and `all` can be used to convert a list of Boolean values into a single Boolean value. The function `any` returns `True` if and only if the list contains the value `True` at least once. The function `all` returns `True` if and only if all values in the list are `True`.

```
>>> a = ['True', 'True']
>>> b = ['True', 'False']
>>> c = ['False', 'False']
>>> d = ['True', 'True', 'True', 'False']
>>> e = ['False', 'True', 'False', 'False']

>>> all(a)
True
>>> all(b)
False
>>> all(d)
False

>>> any(b)
True
>>> any(c)
False
>>> any(e)
True
```

ISBN

The string method `join`

The string method `join` can be used to concatenate all strings in an iterable object (e.g. a `list`) into a single string. This is done by concatenating all strings in the iterable object using a separator, which is the string on which the string method `join` is called.

```
>>> some_list = ['a', 'b', 'c']
>>> ' '.join(some_list)
'a b c'
>>> ''.join(some_list)
'abc'
>>> '---'.join(some_list)
'a---b---c'
>>> ' - '.join(some_list)
'a - b - c'
```

Recoupling

Check if certain conditions hold

Sometimes you need to check explicitly if certain conditions hold when executing part of your program, and the program needs to respond if one of the conditions is not met. One of the easiest ways this can be done in Python is by using an `assert` statement.

```
>>> x = 2
>>> y = 2
>>> assert x == y, 'the values are different'
>>> x = 1
>>> assert x == y, 'the values are different'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: the values are different
```

The general syntax of an `assert` statement is

`assert <condition>, <message>`

The `assert` statement checks whether the condition holds. If this is not the case, an `AssertionError` is raised with the message (`str`) that is given at the end of the `assert` statement. In case this exception is not caught elsewhere in the code (which will always be the case in this course), the execution of the codes halts at the point where the `AssertionError` was raised (*runtime error*).

Traverse the elements of two or more iterable objects simultaneously

If you want to traverse the elements of two or more *iterable objects* (objects of compound data types that have an associated iterator) simultaneously, you do this using the built-in function `zip`. This function returns an iterator that initially returns a tuple containing the first elements of all iterable objects passed to the function `zip`, then a tuple containing all second elements of those iterable objects, and so on.

Say, for example, that you can to add two lists element-wise, thereby creating a new list whose *i*-th element is the sum of the *i*-th elements of the two original lists. This can be done in the following way.

```
>>> first = [1, 2, 3]
>>> second = [4, 5, 6]
>>> added = []
>>> for term1, term2 in zip(first, second):
...     added.append(term1 + term2)
...
>>> added
[5, 7, 9]
```

This can also be written a bit shorter by making use of a *list comprehension*.

```
>>> first = [1, 2, 3]
>>> second = [4, 5, 6]
>>> added = [term1 + term2 for term1, term2 in zip(first, second)]
>>> added
[5, 7, 9]
```

The iterator stops (raises a `StopIteration` exception) as soon as one of the iterable objects is exhausted. If you want to traverse two or more iterable objects simultaneously until the last of those objects is exhausted, you may do this using the function `zip_longest` from the `itertools` module.

Buy 3 get 1 for free

Sorting lists

Python supports two ways to rearrange the elements of a list from the smallest to the largest. You can either call the list method `sort` on the list, or you can pass the list to the built-in function `sorted`. However, there is an important different between these two alternatives. The list method `sort` modifies the list *in place* (and does not return a new list but returns the value `None`), whereas the built-in function `sorted` returns a new list whose elements are sorted from the smallest to the largest.

```
>>> some_list = [4, 2, 3, 1]
>>> some_list.sort()
>>> some_list
```

```
[1, 2, 3, 4]
>>>
>>> some_list = [4, 2, 3, 1]
>>> sorted(some_list)
[1, 2, 3, 4]
```

Sort in descending order

By default the list method `sort` and the built-in function `sorted` sort the elements of a list in ascending order. Both function also have an optional parameter `reverse` to which the value `True` can be passed to have the elements arranged in descending order.

```
>>> some_list = [1, 2, 3, 4]
>>> some_list.sort(reverse=True)
>>> some_list
[1, 2, 3, 4]
>>>
>>> some_list = [4, 2, 3, 1]
>>> sorted(some_list, reverse=True)
[1, 2, 3, 4]
```

Zap reading

Initialize fixed-sized lists with a default value

If you want to create a list with a fixed size `n` whose elements all have the same value `x`, you don't need to use a `for`-loop or a *list comprehension*. The simple solution is to write `[x] * n`.

```
>>> [' '] * 3
[' ', ' ', ' ']
>>> [1] * 5
[1, 1, 1, 1, 1]
```

Note that this multiplication does not make copies of the object `x`, but results in a list whose elements all point to the same object `x`. This is definitely important in case `x` is a *multable* object.

```
>>> some_list = [[1, 2]] * 4
>>> some_list
[[1, 2], [1, 2], [1, 2], [1, 2]]
>>> some_list[0][1] = 666
>>> some_list
[[1, 666], [1, 666], [1, 666], [1, 666]]
>>> some_list[3].append(42)
>>> some_list
[[1, 666, 42], [1, 666, 42], [1, 666, 42], [1, 666, 42]]
```