Complementary sequences

Zipper method

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.

Diffy

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)
```

Energy crisis in New Zealand

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]
```

Not 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.

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

General

Controle of bepaalde voorwaarden gelden

Sometimes it is needed to explicitly check if certain conditions hold when executing part of your source code, 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 the `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 the `assert` statement is

```
assert <condition>, <message>
```

The `assert` statement checks whether or not the condition is met. If this is not the case, an `AssertionError` will be raised with the message 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*).