General

Iterate the positions and the elements of a collection

The built-in function `enumerate` can be used to fetch an iterator for a collection (*iterable*: object of compound data types such as `str`, `list`, `tuple`, files, ...) that both returns the position and the next element in the collection. The example below illustrates how this can be used to simultaneously traverse the characters of a string (`str`) and their positions.

```
>>> for index, character in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'character: {character}')
...
index: 0
character: a
index: 1
character: b
index: 2
character: c
```

This can also be used to simultaneously traverse the characters at corresponding positions of two strings.

```
>>> first = 'abc'
>>> second = 'def'
>>> for index, character in enumerate(first):
...     print(f'{character}-{second[index]}')
...
a-d
b-e
c-f
```

However, in this case it is better to use the built-in function `zip`, which is especially equipped to traverse multiple iterables at once.

```
>>> first = 'abc'
>>> second = 'def'
>>> for character1, character2 in zip(first, second):
...     print(f'{character1}-{character2}')
...
a-d
b-e
c-f
```

Working with *floating point* numbers in doctests

If a function returns a *floating point* number, this might give trouble when testing the correctness of the function using a doctest. This is caused by the fact that doctests perform an exact match between the string that represents the result in the doctest, and the result that is printed or returned by the function. In order to do this, the result of the function is first converted into a string. In comparing these two strings, doctests thus do not take into account the possibility of rounding errors that might occur when working with *floating point* numbers. These rounding errors are a consequence of the limited precision with which computers can represent real-valued numbers.

```
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

If in executing a doctest the expected output (a string) does not exactly match the string representation that is returned by the function, the doctest will consider the result as incorrect.

Dodona does take into account rounding errors when working with *floating point* numbers. Unless otherwise stated in the assignment, Dodona will check if the result is correct up to six decimal digits for functions that return *floating point* numbers (either directly or as elements of collections). This more or less comes down to rewriting a doctest according to the following strategy.

```
def multiply(x):

    """
    >>> abs(multiply(0.1) - 0.3) < 1e-6
    True
    """

    return 3 * x
```

Passing mutable objects to functions

If you pass a mutable object to a function, the function may modify the object *in place*. This might be an explicit goal of the function, but sometimes it is not desirable to modify values that are passed to a function while the function is being executed.

Say, for example, that we pass a list to a function. What we actually pass to the function is a reference to that list and not a copy of the list (*call by reference* instead of *call by value*). As a result, the parameter to which the list is assigned becomes an alias for the list, and the function is able to modify the list itself (after all, lists are mutable data structures).

```
>>> def modify(some_list, element):
...     some_list.append(element)
...     return some_list
...
>>> some_list = ['a', 'b']
>>> modified = modify(some_list, 'c')
>>> modified
['a', 'b', 'c']
>>> some_list
['a', 'b', 'c']
```

In the following example, we first make a copy of the list that is passed to the function. Then we modify the copy, but not the original list. Making a copy of a list can be done for example by using *slicing* (`some_list[:]`) or by using the built-in function `list` (`list(some_list)`).

```
>>> def modify(some_list, element):
...     copy = some_list[:]
...     copy.append(element)
...     return copy
...
>>> some_list = ['a', 'b']
>>> modified = modify(some_list, 'c')
>>> modified
['a', 'b', 'c']
>>> some_list
['a', 'b']
```

The Python Tutor gives a graphical representation of the difference between the above examples:

- example without copying

- example with copying

Because we no longer need the reference to the original list that was passed to the function, we may also rewrite the function `modify` from the above example in the following way.

```
def modify(some_list, element):
    some_list = some_list[:]
    some_list.append(element)
    return some_list
```

In this, the reference of the variable `some_list` to the original list that is passed to the function `modify`, is replaced by a reference to a copy of the list. This replacement is only visible inside the function, because the variable `some_list` is a local variable of the function `modify`. You can also inspect this example using the Python Tutor.

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]
```

Grouping the elements of a list

Python offers multiple solutions for grouping the elements in a list into groups 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 returns an iterator that simultaneously iterates two or more iterables. Each iteration step, the iterator yields a tuple containing the i-th elements of the iterables that are passed to the `zip` function.

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

All and any

The built-in 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
```

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 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*).

The list method `insert`

The list method `append` can be used to add an element to the end of a list. The list method `insert` allows to add an element at a given position in a list. In case the position that is passed to the list method `insert` is greater than or equal to the length of the list, the element is appended at the end of the list.

```
>>> some_list = []
>>> some_list.insert(0, 'a')
>>> some_list
['a']
>>> some_list.insert(0, 'b')
>>> some_list
```

```
['b', 'a']
>>> some_list.insert(1, 'c')
>>> some_list
['b', 'c', 'a']
>>> some_list.insert(10, 'd')
>>> some_list
['b', 'c', 'a', 'd']
```

ISBN

The string method `join`

The string method `join` can be used to concatenate all strings in an iterable (e.g. a `list`) into a single string. This is done by concatenating all strings in the iterable 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'
```

Compilations

Traverse the elements of two or more iterables simultaneously

If you want to traverse the elements of two or more *iterables* (objects of compound data types that have an associated iterator; collections) 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 iterables passed to the function `zip`, then a `tuple` containing all second elements of those iterables, and so on.

Say, for example, that you want 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 iterables is exhausted (raises a `StopIteration` exception). If you want to traverse two or more iterables simultaneously until the last of those objects is exhausted, you may do this using the function `zip_longest` from the `itertools` module.