# General

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

In the example below, 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* (`aList[:]`) or by using the built-in function `list` (`list(aList)`).

```
>>> def modify(aList, element):
...     copy = aList[:]
...     copy.append(element)
...     return copy
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['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 rewrite the function `modify` from the above example in the following way.

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

In this, the reference of the variable `aList` 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 `aList` 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), whereas the built-in function `sorted` returns a new list whose elements are sorted from the smallest to the largest.

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

### Initialize nested list

A rectangular grid with $m$ rows and $n$ columns can be represented by a list `l` containing $m$ lists, with each list from `l` having $n$ elements. Say, for example, that you want to create a list with three rows and two columns, with each cell of the grid containing the empty string. This can be done using *list comprehensions*.

```
>>> m = 3
>>> n = 2
>>> grid = [['' for _ in range(n)] for _ in range(m)]
>>> grid
[['', ''], ['', ''], ['', '']]
>>> grid[0][0] = 'A'
>>> grid
[['A', ''], ['', ''], ['', '']]
```

In most cases you can also use the multiplication operator `*` to create lists of a given size, where each cell is initialized with the same object. But if this object is mutable, you might get surprising results.

```
>>> m = 3
>>> n = 2
>>> [''] * m
['', '', '']
>>> grid = [[''] * n] * m
>>> grid
[['', ''], ['', ''], ['', '']]
>>> grid[0][0] = 'A'
>>> grid
[['A', ''], ['A', ''], ['A', '']]
```

As you can see, putting the string `A` in cell `grid[0][0]`, causes the string `A` to be placed as well in cells `grid[1][0]` and `grid[2][0]`. The reason for this strange behaviour is that the operator `*` has created aliases to the same list, such that `grid[0]`, `grid[1]` and `grid[2]` all reference exactly the same list object. As a result, adjusting one of these lists, changes all of them (well, actually, there is only one of them). This can be seen clearly when you use the Python Tutor.

### 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.

```
>>> aList = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> [(aList[i], aList[i + 1]) for i in range(0, len(aList), 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 (aList[::2]) and the list containing all elements at odd positions (aList[1::2]), you obtain exactly the same result.

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

### String representation of a grid

The following *list comprehension* constructs a string representation of a grid, with each the rows of the grid on a separate line. In other words, the lines are separated from each other by a newline (the string on which the outer `join` method is called). The elements of the rows are separated from each other by a singel space (the string on which the inner `join` method is called).

```
>>> grid = [['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I']]
>>> print('\n'.join([' '.join(rij) for row in grid]))
A B C
D E F
G H I
```

### 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.

```
>>> aList = []
>>> aList.insert(0, 'a')
>>> aList
['a']
>>> aList.insert(0, 'b')
>>> aList
['b', 'a']
>>> aList.insert(1, 'c')
>>> aList
['b', 'c', 'a']
>>> aList.insert(10, 'd')
>>> aList
['b', 'c', 'a', 'd']
```

### 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]]
```