LISTS

CHANGING LISTS

This chapter of our tutorial deals with further aspects of lists. You will learn how to append and insert objects to lists and you will also learn how to delete and remove elements by using 'remove' and 'pop'

A list can be seen as a stack. A stack in computer science is a data structure, which has at least two operations: one which can be used to put or push data on the stack and another one to take away the most upper element of the stack. The way of working can be imagined with a stack of plates. If you need a plate you will usually take the most upper one. The used plates will be put back on the top of the stack after cleaning. If a programming language supports a stack like data structure, it will also supply at least two operations:

- **push**
  This method is used to put a new object on the stack. Depending on the point of view, we say that we "push" the object on top or attach it to the right side. Python doesn't offer - contrary to other programming languages - no method with the name "push", but the method "append" has the same functionality.
- **pop**
  This method returns the most upper element of the stack. The object will be removed from the stack as well.
- **peek**
  Some programming languages provide another method, which can be used to view what is on the top of the stack without removing this element. The Python list class doesn't possess such a method, because it is not needed. A peek can be simulated by accessing the element with the index -1:

```
>>> lst = ["easy", "simple", "cheap", "free"]
>>> lst[-1]
'free'
>>>
```

POP AND APPEND

- s.append(x)
  This method appends an element to the end of the list "s".

```
>>> lst = [3, 5, 7]
>>> lst.append(42)
>>> lst
[3, 5, 7, 42]
>>>
```

  It's import to understand that append returns "None". This means that it usually doesn't make sense to reassign the return value:

```
>>> lst = [3, 5, 7]
>>> lst = lst.append(42)
>>> print(lst)
None
>>>
```

- s.pop(i)
  'pop' returns the ith element of a list s. The element will be removed from the list as well.

```
>>> cities = ["Hamburg", "Linz", "Salzburg", "Vienna"]
>>> cities.pop(0)
'Hamburg'
>>> cities
['Linz', 'Salzburg', 'Vienna']
>>> cities.pop(1)
'Salzburg'
>>> cities
['Linz', 'Vienna']
>>>
```

  The method 'pop' raises an IndexError exception if the list is empty or the index is out of range.
- s.pop()
  The method 'pop' can be called without an argument. In this case, the last element will be returned. So s.pop() is equivalent to s.pop(-1).

```
>>> cities = ["Amsterdam", "The Hague", "Strasbourg"]
>>> cities.pop()
'Strasbourg'
>>> cities
```

```
['Amsterdam', 'The Hague']
>>>
```

EXTEND

We saw that it is easy to append an object to a list. But what about adding more than one element to a list? Maybe, you want to add all the elements of another list to your list. If you use append, the other list will be appended as a sublist, as we can see in the following example:

```
>>> lst = [42,98,77]
>>> lst2 = [8,69]
>>> lst.append(lst2)
>>> lst
[42, 98, 77, [8, 69]]
>>>
```

What we wanted to accomplish is the following:

```
[42, 98, 77, 8, 69]
```

To this purpose, Python provides the method 'extend'. It extends a list by appending all the elements of an iterable like a list, a tuple or a string to a list:

```
>>> lst = [42,98,77]
>>> lst2 = [8,69]
>>> lst.extend(lst2)
>>> lst
[42, 98, 77, 8, 69]
>>>
```

As we have mentioned, the argument of extend doesn't have to be a list. It can be any iterable. This means that we can use tuples and strings as well:

```
>>> lst = ["a", "b", "c"]
>>> programming_language = "Python"
>>> lst.extend(programming_language)
>>> print(lst)
['a', 'b', 'c', 'P', 'y', 't', 'h', 'o', 'n']
```

Now with a tuple:

```
>>> lst = ["Java", "C", "PHP"]
>>> t = ("C#", "Jython", "Python", "IronPython")
>>> lst.extend(t)
```

```
>>> lst
['Java', 'C', 'PHP', 'C#', 'Jython', 'Python', 'IronPython']
>>>
```

EXTENDING AND APPENDING LISTS WITH THE '+'-OPERATOR

There is an alternative to 'append' and 'extend'. '+' can be used to combine lists.

```
>>> level = ["beginner", "intermediate", "advanced"]
>>> other_words = ["novice", "expert"]
>>> level + other_words
['beginner', 'intermediate', 'advanced', 'novice', 'expert']
>>>
```

Be careful. Never ever do the following:

```
>>> L = [3, 4]
>>> L = L + [42]
>>> L
[3, 4, 42]
>>>
```

Even though we get the same result, it is not an alternative to 'append' and 'extend':

```
>>> L = [3, 4]
>>> L.append(42)
>>> L
[3, 4, 42]
>>>
>>>
>>> L = [3, 4]
>>> L.extend([42])
>>> L
[3, 4, 42]
>>>
```

The augmented assignment (+=) is an alternative:

```
>>> L = [3, 4]
>>> L += [42]
>>> L
[3, 4, 42]
```

We will compare in the following example the different approaches and calculate their run times. To understand the following program, you need to know that time.time() returns a float

number, the time in seconds since the so-called „The Epoch"[1]. time.time() - start_time calculates the time in seconds consumed for the for loops:

```
import time

n= 100000

start_time = time.time()
l = []
for i in range(n):
    l = l + [i * 2]
print(time.time() - start_time)

start_time = time.time()
l = []
for i in range(n):
    l += [i * 2]
print(time.time() - start_time)

start_time = time.time()
l = []
for i in range(n):
    l.append(i * 2)
print(time.time() - start_time)
```

This program returns shocking results:

```
26.3175041676
0.0305399894714
0.0207479000092
```

We can see that the "+" operator is about 1268 slower than the append method. The explanation is easy: If we use the append method, we will simply append a further element to the list in each loop pass. Now we come to the first loop, in which we use `l = l + [i * 2]`. The list will be copied in every loop pass. The new element will be added to the copy of the list and result will be reassigned to the variable `l`. After this the old list will have to be removed by Python, because it is not referenced anymore. We can also see that the version with the augmented assignment ("+="), the loop in the middle, is only slightly slower than the version using "append".

REMOVING AN ELEMENT WITH REMOVE

It is possible to remove with the method "remove" a certain value from a list without knowing the position.

```
s.remove(x)
```

This call will remove the first occurrence of x from the list s. If x is not contained in the list, a ValueError will be raised. We will call the remove method three times in the following example to remove the colour "green". As the colour "green" occurrs only twice in the list, we get a ValueError at the third time:

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.remove("green")
>>> colours
['red', 'blue', 'green', 'yellow']
>>> colours.remove("green")
>>> colours
['red', 'blue', 'yellow']
>>> colours.remove("green")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

FIND THE POSITION OF AN ELEMENT IN A LIST

The method "index" can be used to find the position of an element within a list:

```
s.index(x[, i[, j]])
```

It returns the first index of the value x. A ValueError will be raised, if the value is not present. If the optional parameter i is given, the search will start at the index i. If j is also given, the search will stop at position j.

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.index("green")
1
>>> colours.index("green", 2)
3
>>> colours.index("green", 3,4)
3
>>> colours.index("black")
Traceback (most recent call last):
  File "", line 1, in
ValueError: 'black' is not in list
>>>
```

INSERT

We have learned that we can put an element to the end of a list by using the method "append". To work efficiently with a list, we need also a way to add elements to arbitrary positions inside of a list. This can be done with the method "insert":

```
s.insert(index, object)
```

An object "object" will be included in the list "s". "object" will be placed before the element s[index]. s[index] will be "object" and all the other elements will be moved one to the right.

```
>>> lst = ["German is spoken", "in Germany,", "Austria",
"Switzerland"]
>>> lst.insert(3, "and")
>>> lst
['German is spoken', 'in Germany,', 'Austria', 'and', 'Switzerland']
>>>
```

The functionality of the method "append" can be simulated with insert in the following way:

```
>>> abc = ["a","b","c"]
>>> abc.insert(len(abc),"d")
>>> abc
['a', 'b', 'c', 'd']
>>>
```

Footnotes:

[1] Epoch time (also known as Unix time or POSIX time) is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds.