0619 - Leaflet - Functions and methods

July 23, 2018

1 Leaflet: Functions & Methods

1.1 Functions

Functions are combined code blocks. We can use functions to avoid repeating code blocks that are used more than once. Instead, we define a function that contains these code blocks to call the function in further places without copying the lines of code contained in it.

1.1.1 Define and call a function

We already know some functions that Python provides us. The function we have probably used most frequently so far is the print function:

```
In [1]: print("HELLO WORLD")

HELLO WORLD
```

If we want to use our own function, we must define it first. Such a function definition has the general syntax:
def function_name(): **Code**

```
In [2]: def multi_print():
            print("Hello World!")
            print("Hello World!")
```

To execute a function that has been defined, we write: **function_name()**

```
In [3]: multi_print()

Hello World!
Hello World!
```

1.1.2 Functions with an argument

You can pass functions an **argument**, a value on which the code within the function depends.
def function_name(argument): **Code in which the specific argument is used**

```
In [3]: def multi_print2(name):
            print(name)
            print(name)

        multi_print2("Hello")
        multi_print2("World")
```

```
Hello
Hello
World
World
```

You can think of such a parameter as a variable belonging to a function. Avoid naming a function parameter like an existing variable - risk of confusion!

```
In [4]: name = "MARS"

        def multi_print2(name):
            print(name)
            print(name)

        multi_print2("Hello")
        multi_print2("World")

        print(name)
```

```
Hello
Hello
World
World
MARS
```

You see that the value of variable *name* has no effect on the *name* argument of the function!

1.1.3 Further functions in Python

You already know the len function for lists. :-)

```
In [1]: print(len(["Hello", "World"]))
```

```
2
```

You can also use the len function on strings.

```
In [2]: print(len("Hello"))
```

```
5
```

An overview of functions in Python can be found here: https://docs.python.org/3/library/functions.html

1.1.4 Functions with Several Arguments

A function may also contain several arguments.
def Funktionenname(argument1, argument2, ...): Code in which argument1, argument2,... will be worked with

```
In [6]: def multi_print(name, count):
            for i in range(0, count):
                print(name)

        multi_print("Hello!", 5)
```

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

1.1.5 Functions in functions

Functions can also be nested:

```
In [7]: def another_funktion():
            multi_print("Hello!", 3)
            multi_print("World!", 3)
```

```
In [8]: another_funktion()
```

```
Hello!
Hello!
Hello!
World!
World!
World!
```

1.1.6 Return a value

So far, we have been using functions to execute a code block that can depend on arguments. Functions can also return values using the **return** command:

```
In [2]: def return_element(name):
            return name

        print(return_element("Hi"))
```

```
Hi
```

We can then treat such functions with return like variables:

```
In [10]: def return_with_exclamation(name):
             return name + "!"

         if return_with_exclamation("Hi") == "Hi!":
             print("Right!")
         else:
             print("Wrong.")
```

```
Right!
```

```
In [1]: def maximum(a, b):
            if a < b:
                return b
            else:
                return a

        result = maximum(4, 5)
        print(result)
```

```
5
```

Functions vs. methods

1.1.7 Functions

When they are called, functions stand "for themselves" and what they refer to may appear as arguments in the parentheses behind them:

```
In [5]: list1 = [1, 2, 3]
```

```
In [6]: print(list1)
```

```
[1, 2, 3]
```

```
In [7]: print(len(list1))

3
```

1.1.8 Methods

In addition, we already know commands that are appended to objects with a dot. A list is such an **object**. Every object has methods that we can use. However, we cannot apply these methods to an object of another type (usually at least).

Let's look at some useful methods of the list object :-) (you don't have to remember them all)

```
In [8]: # append an element
        list1.append(4)

        print(list1)

[1, 2, 3, 4]
```

```
In [9]: # take an element out of the list at a specific index
        list1.pop(2)

Out[9]: 3
```

```
In [ ]: # we see that the method does not return the updated list, but the removed item
```

```
In [10]: print(list1)

[1, 2, 4]
```

```
In [11]: # Inserting an element at a specific position
         # the first argument at insert specifies which element is inserted into the list
         # the second argument at insert specifies where the element is inserted
         # note that the index of the first element in a list is 0!
         list1.insert(1, 4)

         print(list1)

[1, 4, 2, 4]
```

```
In [12]: # remove an element
         list1.remove(4)

         print(list1)

[1, 2, 4]
```

```
In [14]: # Specify the index of an element (the first position where it occurs)
         print(list1.index(1))
```

```
0
```

```
In [16]: print(list1.index(4))
```

```
2
```

```
In [19]: print(list1.count(4))
```

```
1
```

```
In [20]: # with reverse we can reverse the order of a list
         list1.reverse()
         print(list1)
```

```
[4, 2, 1]
```