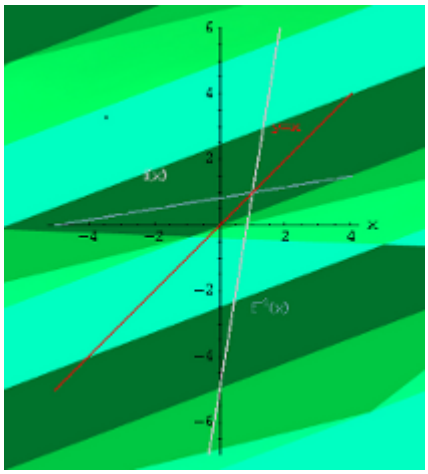
FUNCTIONS

SYNTAX



The concept of a function is one of the most important ones in mathematics. A common usage of functions in computer languages is to implement mathematical functions. Such a function is computing one or more results, which are entirely determined by the parameters passed to it.

In the most general sense, a function is a structuring element in programming languages to group a set of statements so they can be utilized more than once in a program. The only way to accomplish this without functions would be to reuse code by copying it and adapt it to its different context. Using functions usually enhances the comprehensibility and quality of the program. It also lowers the cost for development and maintenance of the software.

Functions are known under various names in programming languages, e.g. as subroutines, routines, procedures, methods, or subprograms.

A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called.
Parameter can be mandatory or optional. The optional parameters (zero or more) must follow the mandatory parameters.

Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value None is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value "None" will be returned.
Example:

```
def fahrenheit(T_in_celsius):
    """ returns the temperature in degrees Fahrenheit """
    return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
    print(t, ": ", fahrenheit(t))
```

The output of this script looks like this:

```
22.6 :  72.68
25.8 :  78.44
27.3 :  81.14
29.8 :  85.64
```

OPTIONAL PARAMETERS

Functions can have optional parameters, also called default parameters. Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used. We will demonstrate the operating principle of default parameters with an example. The following little script, which isn't very useful, greets a person. If no name is given, it will greet everybody:

```
def Hello(name="everybody"):
    """ Greets a person """
    print("Hello " + name + "!")

Hello("Peter")
Hello()
```

The output looks like this:

```
Hello Peter!
Hello everybody!
```

DOCSTRING

The first statement in the body of a function is usually a string, which can be accessed with function_name.__doc__
This statement is called **Docstring**.
Example:

```
def Hello(name="everybody"):
    """ Greets a person """
    print("Hello " + name + "!")

print("The docstring of the function Hello: " + Hello.__doc__)
```

The output:

```
The docstring of the function Hello:  Greets a person
```

KEYWORD PARAMETERS

Using keyword parameters is an alternative way to make function calls. The definition of the function doesn't change.
An example:

```
def sumsub(a, b, c=0, d=0):
    return a - b + c - d

print(sumsub(12,4))
print(sumsub(42,15,d=10))
```

Keyword parameters can only be those, which are not used as positional arguments. We can see the benefit in the example. If we hadn't keyword parameters, the second call to function would have needed all four arguments, even though the c needs just the default value:

```
print(sumsub(42,15,0,10))
```

RETURN VALUES

In our previous examples, we used a return statement in the function sumsub but not in Hello. So, we can see that it is not mandatory to have a return statement. But what will be returned, if we don't explicitly give a return statement. Let's see:

```
def no_return(x,y):
    c = x + y

res = no_return(4,5)
print(res)
```

If we start this little script, None will be printed, i.e. the special value None will be returned by a return-less function. None will also be returned, if we have just a return in a function without an expression:

```
def empty_return(x,y):
    c = x + y
    return

res = empty_return(4,5)
print(res)
```

Otherwise the value of the expression following return will be returned. In the next example 9 will be printed:

```
def return_sum(x,y):
    c = x + y
    return c

res = return_sum(4,5)
print(res)
```

RETURNING MULTIPLE VALUES

A function can return exactly one value, or we should better say one object. An object can be a numerical value, like an integer or a float. But it can also be e.g. a list or a dictionary. So, if we have to return for example 3 integer values, we can return a list or a tuple with these three integer values. This means that we can indirectly return multiple values. The following example, which is calculating the Fibonacci boundary for a positive number, returns a 2-tuple. The first element is the Largest Fibonacci Number smaller than x and the second component is the Smallest Fibonacci Number larger than x. The return value is immediately stored via unpacking into the variables lub and sup:

```
def fib_intervall(x):
    """ returns the largest fibonacci
    number smaller than x and the lowest
    fibonacci number higher than x"""
    if x < 0:
        return -1
    (old,new, lub) = (0,1,0)
    while True:
        if new < x:
            lub = new
            (old,new) = (new,old+new)
        else:
            return (lub, new)

while True:
    x = int(input("Your number: "))
    if x <= 0:
        break
    (lub, sup) = fib_intervall(x)
    print("Largest Fibonacci Number smaller than x: " + str(lub))
    print("Smallest Fibonacci Number larger than x: " + str(sup))
```

LOCAL AND GLOBAL VARIABLES IN FUNCTIONS

Variable names are by default local to the function, in which they get defined.

```
def f():
    print(s)
s = "Python"
f()
```

Output:
Python

```
def f():
    s =
"Perl"
    print(s)

s = "Python"
f()
print(s)
```

Output:
Perl
Python

```
def f():
    print(s)
    s =
"Perl"
    print(s)

s = "Python"
f()
print(s)
```

If we execute the previous script, we get the error message:
`UnboundLocalError: local variable 's' referenced before assignment`
The variable s is ambigious in f(), i.e. in the first print in f() the global s could be used with the value "Python". After this we define a local variable s with the assignment s = "Perl"

```
def f():
    global s
    print(s)
    s =
"dog"
    print(s)
s = "cat"
f()
print(s)
```

We made the variable s global inside of the script on the left side. Therefore anything we do to s inside of the function body of f is done to the global variable s outside of f.

Output:
cat
dog
dog

ARBITRARY NUMBER OF PARAMETERS

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple

reference. This asterisk shouldn't be mistaken with the C syntax, where this notation is connected with pointers.
Example:

```
def arithmetic_mean(first, *values):
    """ This function calculates the arithmetic mean of a non-
empty
        arbitrary number of numerical values """

    return (first + sum(values)) / (1 + len(values))

print(arithmetic_mean(45,32,89,78))
print(arithmetic_mean(8989.8,78787.78,3453,78778.73))
print(arithmetic_mean(45,32))
print(arithmetic_mean(45))
```

Results:

```
61.0
42502.3275
38.5
45.0
```

This is great, but we have still have one problem. You may have a list of numerical values. Like for example

```
x = [3, 5, 9]
```

You cannot call it with

```
arithmetic_mean(x)
```

because "arithmetic_mean" can't cope with a list. Calling it with

```
arithmetic_mean(x[0], x[1], x[2])
```

is cumbersome and above all impossible inside of a program, because list can be of arbitrary length.

The solution is easy. We add a star in front of the x, when we call the function.

```
arithmetic_mean(*x)
```

This will "unpack" or singularize the list.

A practical example:
We have a list

```
my_list = [('a', 232),
           ('b', 343),
           ('c', 543),
           ('d', 23)]
```

We want to turn this list into the following list:

```
[('a', 'b', 'c', 'd'),
 (232, 343, 543, 23)]
```

This can be done by using the *-operator and the zip function in the following way:

```
list(zip(*my_list))
```

ARBITRARY NUMBER OF KEYWORD PARAMETERS

In the previous chapter we demonstrated how to pass an arbitrary number of positional parameters to a function. It is also possible to pass an arbitrary number of keyword parameters to a function. To this purpose, we have to use the double asterisk "**"

```
>>> def f(**kwargs):
...     print(kwargs)
...
>>> f()
{}
>>> f(de="German",en="English",fr="French")
{'fr': 'French', 'de': 'German', 'en': 'English'}
>>>
```

One use case is the following:

```
>>> def f(a,b,x,y):
...     print(a,b,x,y)
…
>>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
>>> f(**d)
('append', 'block', 'extract', 'yes')
```