Table of contents (../toc.ipynb)

Functions

- Functions are very powerful to organize your code and to avoid redundant code.
- To work with functions instead of copy and paste is the first step from spaghetti code to clean code.
- We will talk about clean code later, but one principle is: **A function does one thing!**

Basic syntax

User defined functions follow this syntax.

```
def function_name(argument):
    """Doc string."""
    code
    return something
```

In [25]:
```
# Here a simple function
def my_print(number):
    print("My text", number)
```

In [26]:
```
# Now let's call the function
my_print(3)
```

```
My text 3
```

Arguments

There are

- Required arguments
- Keyword arguments
- Default arguments
- Arguments of variable length

The number in the previous function was a required argument. If you forget it, a syntax error occurs.

In [27]:
```
# Here the required argument is missing
try:
    my_print()  # This does not work
except Exception as error:
    print(error)
```

```
my_print() missing 1 required positional argument: 'number'
```

Here a function with two arguments

In [28]:
```
def my_div(number, denom):
    return number / denom


# And let's call it with the keywords
my_div(number=15., denom=3.)
```

Out[28]:
```
5.0
```

In [29]:
```
# This works also if we flip the keywords
my_div(denom=3., number=15.)
```

Out[29]:
```
5.0
```

Functions with default arguments can either be called with or without the default.

In [30]:
```
def my_scale(num, mult=6):
    return num * mult
```

In [31]:
```
my_scale(3)
```

Out[31]: 18

In [32]:
```
my_scale(3, 2)
```

Out[32]: 6

In [33]:
```
my_scale(mult=3, num=2)
```

Out[33]: 6

Variable length arguments

Here and there, you want to wish to write a function, where you don't know the number of arguments or keyword arguments beforehand.

- The `*args` and `**kwargs` collect all arguments and keyword arguments.
- Every argument which is prefixed with a `*` is converted to a sequence.
- Every argument which is prefixed with two `**` is converted to a dictionary.
- The `*args` and `**kwargs` names are just convention, you can take other names.

In [34]:
```
def print_variable_args(*args, **kwargs):
    print("Here the args", args)
    print("Here the kwargs", kwargs)
```

In [35]:
```
print_variable_args(0, 1, 2, kwarg_one=3, kwarg_two=15)
```

```
Here the args (0, 1, 2)
Here the kwargs {'kwarg_one': 3, 'kwarg_two': 15}
```

In [36]:
```
print_variable_args(0, kwarg_one=3, kwarg_two=15, kwarg_three=-3, )
```

```
Here the args (0,)
Here the kwargs {'kwarg_one': 3, 'kwarg_two': 15, 'kwarg_three': -3}
```

Multiple returns

- Although not very encouraged by clean coding principles, some programmers like to return multiple variables.
- This can be done by `return var1, var2`.
- The return will be a tuple.

In [37]:
```
def mult_return(x, y):
    return x**2, y**2


mult_return(3, 5)
```

Out[37]:
```
(9, 25)
```

As the return becomes a tuple, you can also use indexing to extract just one value.

In [38]:
```
# First return
mult_return(3, 5)[0]
```

Out[38]:
```
9
```

In [39]:
```
# Second return
mult_return(3, 5)[1]
```

Out[39]:
```
25
```

Anonymous functions

- Lambda functions are a nice short hand notation for one liner functions.
- Syntax: `lambda arg, arg: expression`
- They behave differently form functions defined by `def` keyword.
    - They can just return one value.
    - They cannot operate multiple expressions.
    - They cannot access global variables.
- Lambda function are often used if you want to pass a function to a function.

In [40]:
```
sum = lambda x, y: x + y
diff = lambda x, y: x - y

def my_print(x, y, func):
    print(func(x, y))
```

In [41]:
```
my_print(2, 2, func=sum)
```

```
4
```

In [42]:
```
my_print(2, 2, func=diff)
```

```
0
```

However, according to pep8 standard, lambda functions should be avoided and this is easily done with a `def` statement in one line.

In [43]:
```
sum = lambda x, y: x + y
```

In [44]:
```
# can be rewritten as
def sum(x, y): return x + y
```

Type hints

- Starting from Python 3.5, there are type hints which tell other developers or code checkers what kind of variable type a function wants to get.
- Type hints are good for code checkers, code documentation, and development tools.
- Type hints are rather good for large projects with many different developers.
- They are optional.

A function with a type hint looks like:

In [45]:
```
def greeting(name: str) -> str:
    print('Hello ', name)
```

Here, the `(name: str) -> str:` tells that this function wants to see strings. This does not mean that cannot pass "wrong" data type.

In [46]:
```
greeting(0)
```

```
Hello  0
```

In [47]:
```
greeting("Peter")
```

```
Hello  Peter
```

Exercise: Function (10 minutes)

- Write a function, which returns the euclidian distance of the two points: $p = [2, 3, 1]$ and $q = [4, 1, -2]$.
- You should just use two arguments at maximum.

Hint

- The result is 3.

Solution

Please find one possible solution in `solution_function.py` (solution_function.py) file.

In [48]:
```
import sys
sys.path.append("01_basic-python")

from solution_function import *

euclid(p=[2, 3, -1], q=[4, 1, -2])
```

Out[48]:
```
3.0
```