Week 8: Functions

Syntax:

A function is a structuring element in programming languages to group a bunch of statements so they can be utilized in a program more than once. Using functions usually enhances the comprehensibility and quality of a program. It also lowers the cost for development and maintenance of the software.
The following code uses a function with the name *greet*.

```
def greet(name):
    print("Hi " + name)
    print("Nice to see you again!")
    print("Enjoy our video!")

print("Program starts")

greet("Peter")
```

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.

A second example of a function to convert degree to fahrenheit:

```
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))
```

Next, we like to write a function to evaluate the bmi values according to the following table:

Category	BMI (kg/m²)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

The following shows code which is directly calculating the bmi and the evaluation of the bmi, but it is not using functions:

```
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

Ex: Turn the above code into proper functions and function calls.

Default Arguments in Python:

When we define a Python function, we can set a default value to a parameter. If the function is called without the argument, this default value will be assigned to the parameter. This makes a parameter optional. To say it in other words: Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

Run the following code snippet:

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

hello("Peter")
hello()
```

However, one needs to be careful about mutable objects in python. Mutable objects are those which can be changed after creation. In Python, dictionaries are examples of mutable objects. Passing mutable lists or dictionaries as default arguments to a function can have unforeseen effects. Programmers who use lists or dictionaries as default arguments to a function, expect the program to create a new list or dictionary every time that the function is called. However, this is not what actually happens. Default values will not be created when a function is called. Default values are created exactly once, when the function is defined, i.e. at compile-time.

Let us look at the following Python function "spammer" which is capable of creating a "bag" full of spam:

```
def spammer(bag=[]):
    bag.append("spam")
    return bag
```

Call this function say 5 times. What do you get?

To understand what is going on, you have to know what happens when the function is defined. The compiler creates an attribute __defaults__:

```
def spammer(bag=[]):
    bag.append("spam")
    return bag

spammer.__defaults__
```

Whenever we will call the function, the parameter bag will be assigned to the list object referenced by `spammer.__defaults__[0]`
Run the following code snippet:

```
for i in range(5):
    print(spammer())

print("spammer.__defaults__", spammer.__defaults__)
```

Now, you know and understand what is going on, but you may ask yourself how to overcome this problem. The solution consists in using the immutable value `None` as the default. This way, the function can set bag dynamically (at run-time) to an empty list:

```
def spammer(bag=None):
    if bag is None:
        bag = []
    bag.append("spam")
    return bag

for i in range(5):
    print(spammer())

print("spammer.__defaults__", spammer.__defaults__)
```

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 had keyword parameters, the second call to function would have needed all four arguments, even though the c argument 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. It is not mandatory to have a return statement, however. What would 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)
```

The same 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)
```

Function bodies can contain one or more return statements. 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.

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. That is, 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. Run the following code snippet and check the output.

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

while True:
    x = int(input("Your number: "))
    if x <= 0:
        break
    lub, sup = fib_interval(x)
```

```
    print("Largest Fibonacci Number smaller than x: " + str(lub))
    print("Smallest Fibonacci Number larger than x: " + str(sup))
```

Returning a list

It is also possible to define functions to return a list of values. Say we have a sequence $x_n = (1 + \frac{1}{n})^n$ and we want to obtain the first n values of the sequence, where n is provided as an input. We can do this using the following function.

```
def sequence(n):
      values = []
      for i in range(1,n+1):
            values.append((1+ 1./i)**i)
      return values
sequence(5)
```

```
[2.0, 2.25, 2.37037037037037, 2.44140625, 2.4883199999999994]
```

Note that `range(n)` is the same as `range(0,n)` which returns a list starting from 0(inclusive) to n-1(inclusive). `range()` itself is an inbuilt function which takes integers as arguments and returns a list.

Try yourself: Generate the sequence for large values of n and observe if it converges to e

Global and local variables in functions

In the following example, we want to demonstrate, how global values can be used inside the body of a function:

```
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

```
I love Paris in the summer!
```

The variable s is defined as the string "I love Paris in the summer!", before calling the function f(). The body of f() consists solely of the "print(s)" statement. As there is no local variable s, i.e. no assignment to s, the value from the global variable s will be used. So the output will be the string "I love Paris in the summer!". The question is, what will happen if we change the value of s inside of the function f()? Will it affect the global variable as well? We test this in the following piece of code:

```
def f():
    s = "I love London!"
    print(s)
```

```
s = "I love Paris!"
f()
print(s)
```

```
I love London!
I love Paris!
```

What if we combine the first example with the second one, i.e. we first access s with a print() function, hoping to get the global value, and then assigning a new value to it? Assigning a value to it, means - as we have previously stated - creating a local variable s. So, we would have s both as a global and a local variable in the same scope, i.e. the body of the function. Python fortunately doesn't allow this ambiguity. So, it will raise an error, as we can see in the following example:

```
def f():
    print(s)
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-3-d7a23bc83c27> in <module>
      5
      6 s = "I love Paris!"
----> 7 f()

<ipython-input-3-d7a23bc83c27> in f()
      1 def f():
----> 2     print(s)
      3     s = "I love London!"
      4     print(s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

A variable can't be both local and global inside a function. So Python decides that we want a local variable due to the assignment to s inside f(), so the first print statement before the definition of s throws the error message above. Any variable which is changed or created inside a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global", as can be seen in the following example:

```
def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)


s = "I am looking for a course in Paris!"
f()
```

```
print(s)

I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

Local variables of functions can't be accessed from outside when the function call has finished. Here is the continuation of the previous example:

```
def f():
    s = "I am globally not known"
    print(s)

f()
print(s)

I am globally not known
Only in spring, but London is great as well!
```

The following example shows a wild combination of local and global variables and function parameters:

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)

42 17 4 17
42 15 3 4
```

Global variables in nested functions

We will examine now what will happen, if we use the global keyword inside nested functions. The following example shows a situation where a variable 'city' is used in various scopes:

```
def f():
    city = "Hamburg"
    def g():
        global city
        city = "Geneva"
```

```
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

f()
print("Value of city in main: " + city)
```

```
Before calling g: Hamburg
Calling g now:
After calling g: Hamburg
Value of city in main: Geneva
```

We can see that the global statement inside the nested function g does not affect the variable 'city' of the function f, i.e. it keeps its value 'Hamburg'. We can also deduce from this example that after calling f() a variable 'city' exists in the module namespace and has the value 'Geneva'. This means that the global keyword in nested functions does not affect the namespace of their enclosing namespace! This is consistent to what we have found out in the previous subchapter: A variable defined inside of a function is local unless it is explicitly marked as global. In other words, we can refer to a variable name in any enclosing scope, but we can only rebind variable names in the local scope by assigning it to it or in the module-global scope by using a global declaration. We need a way to access variables of other scopes as well. The way to do this are nonlocal definitions, which we will explain in the next chapter.

nonlocal variables

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. One difference to global variables lies in the fact that it is not possible to change variables from the module scope, i.e. variables which are not defined inside of a function, by using the nonlocal statement. We show this in the two following examples:

```
def f():
    global city
    print(city)

city = "Frankfurt"
f()
```

```
Frankfurt
```

This program is correct and returns 'Frankfurt' as the output. We will change "global" to "nonlocal" in the following program:

```
def f():
    nonlocal city
    print(city)

city = "Frankfurt"
f()
```

```
 File "<ipython-input-9-97bb311dfb80>", line 2
    nonlocal city
```

```
    ^
SyntaxError: no binding for nonlocal 'city' found
```

This shows that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

```
def f():
    city = "Munich"
    def g():
        nonlocal city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
Before calling g: Munich
Calling g now:
After calling g: Zurich
'city' in main: Stuttgart
```

In the previous example the variable 'city' was defined prior to the call of g. We get an error if it isn't defined:

```
def f():
    #city = "Munich"
    def g():
        nonlocal city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
 File "<ipython-input-11-5417be93b6a6>", line 4
    nonlocal city
    ^
SyntaxError: no binding for nonlocal 'city' found
```

The program works fine - with or without the line 'city = "Munich"' inside of f - , if we change "nonlocal" to "global":

```
def f():
```

```
    #city = "Munich"
    def g():
        global city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
Before calling g: Stuttgart
Calling g now:
After calling g: Zurich
'city' in main: Zurich
```

Exercises for submission. Create a notebook and solve the following. Try to solve each exercise in a single cell.

1. A leap year is a calendar year containing an additional day added to keep the calendar year synchronized with the astronomical or seasonal year. In the Gregorian calendar, each leap year has 366 days instead of 365, by extending February to 29 days rather than the common 28. These extra days occur in years which are multiples of four (with the exception of centennial years not divisible by 400). Write a Python function, which asks for a year and calculates if this year is a leap year or not.
2. Write a function which takes a text and encrypts it with a Caesar cipher. This is one of the simplest and most commonly known encryption techniques. Each letter in the text is replaced by a letter with a fixed number of positions further in the alphabet. The function takes two input arguments; the first input argument is the message to be encrypted, the second input argument is the shift argument which is the

number of places every character is shifted. For example if the shift argument is 3, then "cat" is encrypted as "fdw" and "say" is encrypted as "vdb". Assume that the message to be encrypted contains only alphabets.

3. Perhaps the first algorithm used for approximating the square root of a number is known as the "Babylonian method". If a number x_n is close to the square root of a, then $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ will be a better approximation. Write a function to calculate the square root of a number by using the Babylonian method.

   The function should take three arguments: the first argument is the number a whose square root we want to find; the second argument is the initial x_0 ; the third argument is the ϵ argument which is used as a termination criteria in the following way. Say ϵ is set at 0.001. After a certain number of iterations of the Babylonian method we stop iterating once the condition $x_{n+1} - x_n \vee 0.001$is satisfied.

   The function should return two values: the first value is the square root of the number provided as the input argument. The second value is the number of iterations it took to terminate for a given ϵ.

4. In exercise 3, fix an a and find the number of iterations it takes to terminate for different values of epsilon (take at least 5 different values of epsilon). Plot the number of iterations vs epsilon. What is your observation?

5. Write a function fuzzy_time which expects a time string in the form hh: mm (e.g. "12:25", "05:35"). The function rounds up or down to a quarter of an hour. Examples: fuzzy_time("12:58") ---> "13:00", fuzzy_time("12:25") ---> "12:30", fuzzy_time("05:47") ---> "05:45".

6. Write a function that takes a list of numbers as an argument and returns the median of the list of numbers.

7. Write a function prime_list(n) that takes a natural number n as an argument and returns the list of first n prime numbers. For example, prime_list(1) should return [2], prime_list(3) should return [2,3,5]. Test your function for n = 11.