5 Functions

Functions are named sequences of statements which enable repeated execution of a block of code with different input parameters. To execute a function, a **function call** (using a name of the function in a script) is needed. We have already seen a few function calls: `print("Hello world")`, `type(12.0)` or mathematical function `abs(-5)`. The expression in parentheses is called the **argument** of the function. It is common to say that a function **takes** an argument and **returns** a result. The output of the function (a result) is called the **return value**. Each function should have one specific job – that improves the modularity and reusability of the code.

Why should you write functions?

- creating a function enables giving a group of statements, that are part of the solution, a name which makes a program easier to read (code is organised better),
- repetitive code can be eliminated and a program may become smaller, any changes need to be done only in one place instead of each repetition (minimizing redundancy),
- well-designed functions can be useful to more than one program (code reuse);
- dividing a long program into functions makes it easier to debug one part at a time,
- coding a function in a file enables importing the file as a module and using it in any other program.

5.1 Function definition

A definition of a function in Python is stated with `def` keyword followed by a name of the function, a parenthesis and a colon with at leas one expression in a indented block of code (example 5.1). When Python reaches and executes a `def` statement, it generates a new **function object** and assigns it to the function's name. The function name becomes a reference to the function object in the same way as a variable name is a reference to a variable. Function name can be assigned to other names, stored in a list or passed to another function as an argument.

Example 5.1. Definition of a function

```
1 def function_name():
2     pass
```

The rules for function names are the same as for variable names – they have to start with a letter or an underscore and contain only letters, numbers or underscores. **A function should not have the same name as any variable** used in the script to avoid confusion and overwriting already existing values. In parentheses the parameters are included. Empty parentheses after function name indicate that a function does not take any input values. First line of the function is called a **header** and the header has to end with a colon (`:`). Then, the sequence of statements – the **body**, which runs when a function is called, is specified. The body has to be an indented block of code. A `return` statement is an exit point from a function (example 5.2).

Example 5.2. A function with parameters and a return statement

```
1 def function_name(parameter1, parameter2, parameter3 = "default_value"): # function header
2     # instructions (body of a function)
3     return returned_value
```

`def` is an executable statement, thus, it can appear anywhere a statement can – even nested in other statements. `def`s are not evaluated until they are reached and run (Python is not a compiled language, so functions are not precompiled). The instructions inside the function are not executed until the function

is **called** and the function definition itself generates no output. Like a variable, a function name must be assigned to a function object before it can be used – a function definition has to appear in the code structure (or run in case of Jupyter Notebook) before the function call. You call a function by typing its name and parentheses (with arguments inside, if a function takes any)

Not all the functions return results. Those that do not have a `return` statement and simply perform an action are called **void functions**. When a function do not return a value explicitly, it **returns the null object** (`None`). The opposite of void functions are **fruitful functions** (return results which are not null values, unless defined as null). If a fruitful function is called and the returned value is not assigned to any variable, the result value is lost forever (important when running in script mode, interactive mode displays the results, see example `[1]`). If the result of the void function is assigned, a `None` value will be obtained. The value `None` has its own type (`NoneType`).

In [1]:
```
def square(x):
    return pow(x, 2)

x = 2
x_squared = square(x) # the result of calling square with value x is stored in a new
     variable
square(3) # the result of squaring 3 is not stored, it's lost

#result of a void function
print(print("Hello")) # will print "Hello" (function call) and None -> the result of
     a printing print("Hello")
```

```
Hello
None
```

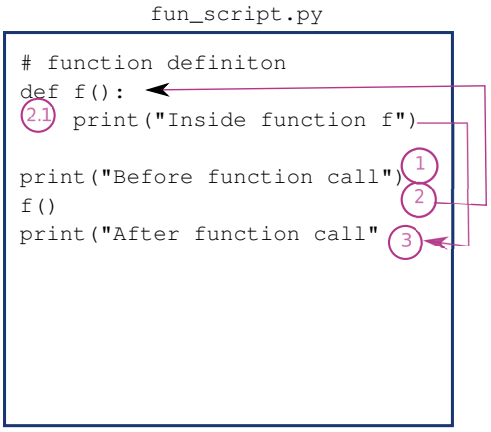

Figure 5.1. Flow of the script with a function

A function needs to be defined in order to be executed without errors. The **flow of execution** defines an order in which statements in script run. Execution always begins at the first statement of the program. Statements are run from top to bottom, one at a time. Function definitions do not change the flow of the program, but statements inside a function are not run, if a function is not called. Figure 5.1 demonstrates the flow of execution. Despite the fact that the function was defines first. The function is run through, a function object is created, but there is no effect of the function run until the function is called. The first statement which is executed and which results is visible is step 1. A function call (step 2) disturbs the flow of the execution in a way that instead of going to a next statement, the flow jumps to the body of the function, runs the instructions there (step 2.1) and comes back to where it left off (Step 3). A

function can call another function, but Python is good at keeping track of where to come back. When it gets to the end of the program, it terminates.

5.2 Parameters and arguments

The values passed into a function when it is called are known as **arguments**. When a function is called with arguments, theirs vales are assigned to their corresponding **parameters** inside the function. In other words, parameter is a name inside a function used to refer to a value passed as an argument. In example `[2]`, the function assigns values of the arguments (passed in line 5) to parameters called `x` and `y` and prints values of those parameters.

Some of the functions require arguments, sometimes more than one. A minimal parameter number is zero (like void functions in C, defined with empty parentheses).

In [2]:
```
1 def print_arguments_function(x, y):
2     print(x)
3     print(y)
4
5 print_arguments(5, 6.8)
```

```
5
6.8
```

An argument in a function call can be any kind of expression including arithmetic operations or even a function object or a function call (example `[3]`).

In [3]:
```
1 import math
2
3 result = math.sin(90 * math.pi / 180)
4 x = abs(math.sqrt(9)) # using a function call -> result of sqrt as input to abs
      function
5 y = 8
6 example_function(x, y)
```

```
3.0
8
```

The name of the variable passed as an argument has nothing to do with the name of the parameter (`x` and `y` in example `[2]`). A variable inside a function will not overwrite the variable outside of function, because it is not visible outside of the function (see subsection 5.4).

Arguments are passed by object reference – the parameter becomes a new reference to the already existing object. For immutable objects, what is done with the parameter has no effect outside of the function. If a mutable object, e.g. a list, is passed to a function and then modified (in place), the changes will also appear in the calling program – they are permanent. Python is passing the actual object, not its copy. To send a copy of a list to a function and prevent from modifying it, use the code in example `[4]`. The slice notation creates a copy of a list and sends it to a function.

In [4]:
```
def substitute_list(list_1):
    list_1[0] = 1

l_1 = [10, 9, 8, 7]
print("Before", l_1)
substitute_list(l_1[:])
print("After (with colon)", l_1)
substitute_list(l_1)
print("After (without colon)", l_1)
```

```
Before [10, 9, 8, 7]
After (without colon) [1, 9, 8, 7]
After (with colon) [1, 9, 8, 7]
```

Reassigning the parameter does not affect the argument outside of the function (example `[5]`). As soon as a function executes the assignment of the argument, the reference is rebound, and the connection to the original object is lost. However, the changes can be still done inside of a mutable object, reassigning one of the elements of a list or a dictionary will have an effect outside the function. The described example is a side effect, an unexpected behaviour of the language that modifies the calling environment. We can avoid this kind of unwanted effect using the `return` statement correctly.

In [5]:
```
def modify_lists(list_1, list_2, dictionary):
    list_1.append("new")
    dictionary[1] = 'a'
    list_2 = [2, 3, 4]
    # notice the changes in list_1 and no changes in list_2 due to reassignment

l1 = ["a", "b"]
l2 = [1]
dictionary = {1: "z", 3: "zzz", 2: "zz"}
print("Before", l1, l2, dictionary)
modify_lists(l1, l2, dictionary)
print("After", l1, l2, dictionary)
```

```
Before ['a', 'b'] [1] {1: 'z', 3: 'zzz', 2: 'zz'}
After ['a', 'b', 'new'] [1] {1: 'a', 3: 'zzz', 2: 'zz'}
```

Any Python object, including a function, can be used as a function argument. Arguments may be given default values, then their call in a function is optional. Arguments are passed by position (unless stated otherwise) – values passed in a function call match parameter names in a function definition from left to right. **Positional arguments** are arguments that need to be in the same order that they are written. Python matches values provided in the function call with arguments in the same order they were provided. **Keyword arguments** consist of a name of the variable and its value (a pair of name and value). The order in which the arguments appear is not important because the keywords are used to assign values to parameters (example `[6]`).

In [6]:
```
def plus(x, y):
    print(f"{x=}, {y=}")
    return x + y

a, b = 1, 2
result_1 = plus(a, b)   # positional arguments
result_2 = plus(y=a, x=a) # keyword arguments, notice the order is switched
```

```
x=1, y=2
x=1, y=1
```

Default values can be defined for each parameter. If an argument value is provided in the function call, Python uses the given value to evaluate the function. Otherwise, the default value is used (example [7]). Using default values can simplify the function calls or suggest in which ways is the function typically used. Parameters with default values have to be defined after all of the parameters that do not have the default value. This order allows Python to interpret positional parameters correctly.

In [7]:
```
import math as m
def N(phi, a=6378137, e2=0.0066943802):
    """
    default ellipsoid: GRS-80
    a - major axis
    e2 - eccentricity squared
    """
    phi = m.radians(phi) # switch phi to radians before calculating sine function
    return a / (m.sqrt(1-e2*m.sin(phi)**2))

# GRS-80 ellipsoid, using default values
N_grs = N(52)

# Krasowski ellipsoid, providing values other than default
N_k = N(52, a=6378245, e2=0.0066934216)
print(f"{N_grs=}, {N_k=}")
```

```
N_grs=6391435.268629482, N_k=6391541.583573859
```

Default argument values are calculated when a function is defined, not when it is called. A common error is to use a mutable data type as a default argument (see example [8]). In example [8], the list `b` is expected to be empty each time a function is called with only `a` argument. However, the list `b` will be empty only the first time that the function `modify_list` is called, the next time, list `b` will contain the items from the previous calls when no `b` was given.

In [8]:
```
def modify_list(a, b=[]):
    b.append(a) # each function call, value of a will be attached to the list b,
        previous values of b will be kept
    return b

z = modify_list(5)
z = modify_list(10)
z = modify_list('a') # z will contain elements 5, 10, 'a'
y = modify_list(5, []) # the value will be added to the empty list
print(f"{z=}, {y=}")
```

```
z=[5, 10, 'a'], y=[5]
```

If you need to use a mutable type in your function, unless your goal is to modify it during each function call, it is best to substitute it for `None` and define an empty mutable value in the body of a function, like in example `[9]`.

In [9]:
```
def modify_list(a, b=None):
    if b is None: # if b was not defined in a function call, we reset its value to an
         empty list
        b = []
    b.append(a) # each function call, appends a value to an empty list or a list
        defined by the user
    return b

z = modify_list(5)
z = modify_list(10)
z = modify_list('a') # z will contain only 'a'
y = modify_list(18, [])
print(f"{z=}, {y=}")
```

```
z=['a'], y=[18]
```

The number of input parameters may change depending on the usecase. A parameter name beginning with * gathers arguments into a tuple of parameter values and enables passing a different number of values in each function call. In example `[10]`, `x` and `y` are the positional parameters and `args` is a tuple of arbitrary length, containing the **excess parameters** (also called **arbitrary arguments**). The asterisk before its name suggests that `args` is an arbitrary argument. If calling the function `function_name` with four parameters as in example `[10]`, `x` will have a value of `a`, `y` will have a value of `b` and `args` will contain `(c, d, e)` grouped into a tuple.

The positional parameters must always be listed before the excess parameters. When you are using * (arbitrary arguments), you do not need to call your parameter `args`, but is common to do so.

In [10]:
```
def function_name(x, y, *args):
    list_a = []
    list_a.extend([x, y])
    # process the rest of elements, if there are any
    for item in args:
        list_a.append(item)
    return list_a

# calling function function_name:
a, b, c, d, e = 3, 4, 5, 1, 2
result = function_name(a, b, c, d, e)
print(result)
```

```
[3, 4, 5, 1, 2]
```

Asterisk have one more use. To pass a sequence of a parameters as individual values, the * operator is used to scatter the sequence (example `[11]`) – kind of like tuple unpacking, opposite of what is done with grouping in a function call. This type of unpacking is called **tuple unpacking**, however, it can be applied in a function call to other iterables like lists or sets.

In [11]:
```
def plus(x, y):
    return x + y

a = [1, 2]
result = plus(*a)
print(result)
```

```
3
```

Functions can also be written in a way that they accept as many key-value pairs as the function call provides. This approach is used when it is not clear ahead of time, what kind of information will be provided to a function. Arbitrary keyword arguments are defined in a function header with double asterisk, which creates an empty dictionary and pack name-value pairs into a dictionary, and given in a function call as `key=value` pairs (example [12]). Those arguments are treated as a dictionary inside a function. As with `*args`, the dictionary of arbitrary arguments does not have to be named `kwargs`, but it is common usage (in example [12] the dictionary is called `family_members`).

In [12]:
```
def family(first_name, last_name, **family_members):
    print(first_name+' '+last_name+"'s family members include")
    for key, value in family_members.items():
        print(key, ":", value)

family("John", "Smith", brother="Walter", sister="Agnes", mother="Samantha", father=
    "Adam")
```

```
John Smith's family members include
brother : Walter
sister : Agnes
mother : Samantha
father : Adam
```

It is possible to mix all the above-mentioned types of passing the arguments, although a certain order must be kept. The positional arguments come first, then named arguments (default values), next come the indefinite positional argument with a single asterisk (`*`) and lastly, the indefinite keyword arguments introduced with `**`.

Starting from version 3.8, it is possible to define which parameters can be given as positional only or keyword only. Two seperators in the parameters list were introduced: `/` and `*`. `/` divides the positional only parameters from the rest (positional or keyword and keyword) and `*` indicates that there are no more positional parameters and starting from the right of the symbol, only keyword parameters can be given (example 5.3 and [13]). Positional-only parameters cannot be specified by keyword.

Example 5.3. Dividing positional only and keyword only parameters

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
#     -----------    ----------     ----------
#     ^              ^                    ^
#     |              |                    |
#     |              Positional or keyword |
#     |                                    -- Keyword only
#      -- Positional only
    pass
```

In [13]:
```
def f(x, y, /, z):
    print(f'x: {x}')
    print(f'y: {y}')
    print(f'z: {z}')
```

In [14]:
```
f(1, 2, 3) # valid call
```

```
x: 1
y: 2
z: 3
```

In [15]:
```
f(1, 2, z=4) # valid call
```

```
x: 1
y: 2
z: 4
```

In [16]:
```
f(x=1, y=2, z=3) # invalid call
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-ae17ac4cb3d4> in <module>
----> 1 f(x=1, y=2, z=3)

TypeError: f() got some positional-only arguments passed as keyword arguments: 'x,y'
```

In example [17], both positional-only and keyword-only designators are used.

In [17]:
```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

f(1, 2, c=0, d=1, e="e", f="f")
f("a", "b", "c", d="d", e="e", f="f")
```

```
1 2 0 1 e f
a b c d e f
```

5.3 Return values

`return` statement included in a function can be interpreted as "return immediately from this function and use the following expression as a return value". The expression does not have to be a single variable, it can be a mathematical expression, like `4 * a`. On the other hand, **temporary variables** used to store computational results can make debugging easier.

A function can include more than one `return` statement, e.g., one for each branch of a conditional (example [18]). A `return` statement can be included anywhere in a body of a function. As soon as return statement runs, the function terminates and none of the subsequent statements is executed. It is a good idea to ensure that every possible path through the program reaches a `return` statement – if a program reaches an end of a a function without a `return`, the return value is `None`. A `return` statement without a value also returns to the caller and sends back a `None` value.

In [18]:
```
def sign_x(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1

sign_x(8), sign_x(-1) # call the function for two values (creates a tuple)
```

Out[18]:
```
(1, -1)
```

A function can return only one value, but if you specify multiple values separated with a comma in a `return` statement, the values are packed and returned as a tuple. Returned values can be assigned to individual variables using a tuple unpacking (example [19]).

In [19]:
```
def fmax_min(x):
    return max(x), min(x)

max_a, min_a = fmax_min(a)
print(max_a, min_a)
```

```
2 1
```

Functions can return booleans, which are often used for writing tests in a form of a function. It is a common practice to give boolean function a name sounding like a yes/no question: `is_alphanumeric`, `is_alpha`. Boolean functions can be used in conditional statements. The comparison with a boolean value is unnecessary (example [20]).

In [20]:
```
def is_positive(x):
    if x > 0:
        return True
    else:
        return False

x = 8
if is_positive(x):      # instead of is_positive(x) == True, result of the function
    does not need to be assigned to a variable
    print("x is positive")
else:
    print("x is not positive")
```

```
x is positive
```

5.4 Variable scope

Scope, the place where variables are defined and looked up, prevent name clashes across the program's code. Scope can provide a way to retain state information between function calls. Scope determines name's visibility to the code.

Namespace is the dictionary that contains the names of the variables and their values, a place where names "live". Namespaces are created and updated automatically during the program run.

Three levels of namespaces in Python can be distinguished:

1. **local namespace** – created when a function is called; it contains the argument values assigned to parameter names and all the other variables that are created within a function. When a function terminates, local variables are deleted. A variable created inside a function is not visible outside the function, its scope is only the function's local namespace. Parameters are also local. Each call to a function creates a new local scope.

2. **global namespace** – created when a module is loaded. Each module has its own namespace. Variables assigned in a global scope are visible to any function within the module – if a variable is assigned outside all `defs`, it is **global** to the entire file. Global scope spans a single file only.

3. **built-in namespace** – created when the interpreter starts running the program. It contains the functions that comes with the Python interpreter, which can be accessed by any program unit.

During an execution of a function, variable name encountered is resolved by searching the scopes in order: (1) local namespace, (2) global namespace, (3) built-in namespace. If the name cannot be resolved, a `NameError` exception is raised.

Global scope is visible to functions within the module, it is not necessary to pass them as arguments (it is a good practice, though).

5.5 Recursion

We have seen that a function can call another function. But it is also legal for a function to call itself. A function that calls itself is **recursive** and a process of executing it is called **recursion**. The recursion is usually used to call a function with a decreasing value of an argument – a problem is simplified to two steps: a base case (works as an endpoint for recursion) and modification of a value obtained from a recursive call.

In [21]:
```
def power_rec(a, n):
    if n == 1:       # the most elementary operation
        return a
    else:
        return a * power_rec(a, n-1)

power_rec(2, 5)
```

Out[21]:
```
32
```

After recursion reaches the **base case** (no more recursive calls), the flow of execution returns to the caller function and the remaining lines do not run. If a recursive function never reaches a base case, the program never terminates. This problem is known as **infinite recursion** and is not a recommended approach.

The recursive functions can be usually written with a loop instead of a recursive way, but it is not always possible.

5.6 Anonymous functions

The `lambda` keyword creates an anonymous function within a Python expression. Syntax limits the body of a `lambda` function to a single expression (example `[22]` and `[23]`). `lambda` cannot make assignments or use more complex syntax as `while`, `if`, `try` etc. They do not have a `return` statement, a value is automatically returned.

`Lambda` expression creates a function object in the same way as `def` statement does, but they do not have to have names (they are called anonymous functions for a reason).

In [22]:
```
# immediate invoking
(lambda x, y: x + y)(1, 2) # immediately evaluates the expression for x=1 and y=2
```

Out[22]:
```
3
```

In [23]:
```
# naming lambda for later use
power_x_to_y = lambda x, y: x**y

a = power_x_to_y(2, 4)
print(a)
```

```
16
```

`Lambda` functions can be defined in the same place in the code they are used (inline), instead of separate definition in a different place from where it's used. `Lambda` functions are useful for higher-order functions[1] (example `[24]`): `map`, `reduce`, `sort`, `sorted`, `max`, `min`, etc.

In [24]:
```
# sort elements in a list based on the reverse order of the two last characters
fruits = ["cherry", "apple", "orange", "pear", "plum", "banana"]
print(sorted(fruits)) # lexicographic sort: ["apple", "banana", "cherry", "orange",
    "pear", "plum"]
sorted_last2 = sorted(fruits, key=lambda x: x[:-3:-1])
print(sorted_last2)  # ["banana", "orange", "apple", "plum", "pear", "cherry"]
```

```
['apple', 'banana', 'cherry', 'orange', 'pear', 'plum']
['banana', 'orange', 'apple', 'plum', 'pear', 'cherry']
```

5.7 Documenting functions

To improve code readability, a few rules should be taken into account. Names of the functions should be descriptive – to help understand what the code is trying to do. The names should use lowercase characters and underscores (module names should follow that convention as well).

5.7.1 Docstring

Every function should have a comment explaining what a function does. This comment is called a **docstring** (documentation string). Docstring is placed immediately after the function header and is supposed to follow a docstring format (described in PEP-257). Apart from the information containing a synopsis of the function, it should describe the parameters that the function takes, a value that is returned, side effects and possible exceptions raised (example `[25]`). You can also include a reference to a method used in a function (for your future self in case you forget what the algorithm was). A docstring can be accessed using `help(function_name)` function or by calling `function_name.__doc__`. `PEP-257` does not really specify how the docstring should look like, IDEs and even different Python modules follow few conventions. An example of reST/Sphinx style is included in example `[25]`, some other templates can be found in `https://stackoverflow.com/questions/3898572/what-is-the-standard-python-docstring-format`.

According to PEP-8, No spaces should be used on either side of an equal side when specifying a default value for a parameter or in keyword arguments. If a module has more than one function, separate them with two blank lines. All import statements should be included at the beginning of a file.

[1] functions which take other functions as arguments

In [25]:
```
def function_example(x, y):
    """
    This is a short function description. reST/Sphinx style used below.

    :param x: description of what is x used for
    :param y: description of what is y used for
    :returns: descriptionof what is returned
    """
    return x**y

help(function_example)
```

```
Help on function function_example in module __main__:

function_example(x, y)
This is a short function description. reST/Sphinx style used below.

:param x: description of what is x used for
:param y: description of what is y used for
:returns: descriptionof what is returned
```

5.7.2 Annotations

Python 3 provides also a mechanism called `function annotations` – syntax to attach metadata to the function parameters declaration and the return value (example [26]). Each argument of a function may have an annotation defined with a colon (`:`). The return value is annotated with `->`. The most common types used in annotations are classes (int, str, float) or strings like `"int >0"`. Annotations are stored in `__annotations__` attribute of a function.

In [26]:
```
def power_x_to_y(x:float, y: "int > 0") -> float:
    return x**y

power_x_to_y.__annotations__
```

Out[26]:
```
{'x': float, 'y': 'int > 0', 'return': float}
```

However, annotations have no semantic meaning to the interpreter – no checks or validations are done when the code runs, although it is possible to specify some tests using annotations. They make good documentation, though, and allow to store some additional metadata for functions, like description of variables, their types or expected units.

5.8 Exercises

Exercise 5.1. Write a function calculating an area of a circle, given its radius.

Exercise 5.2. Move the definition of a function after the function call. Run the program to see what error messages you get.

For this exercise you can use the function from the previous exercise, but make sure to restart the kernel of the Notebook first, to clear all the existing variables.

You can also define a new function (area of a rectangle? addtion of two values?), but, also in this case, remember to restart the kernel first.

Exercise 5.3. Write a function computing an area of an arbitrary triangle given the lengths of its edges. Write a function which computes area of a triangle, given its vertices. Remember to check, if the edges create a triangle.

Exercise 5.4. Write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

Exercise 5.5. Write a function `diff(f, x, h=1E-6)` for numerical differentiation:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

that returns an approximation of a derivative of a function f (f should be defined in Python).

Exercise 5.6. Write a function returning the maximum value in a sequence. Try to solve this task without using `max` function.

Exercise 5.7. Write a function returning the minimal value among the given values. Number of values can be arbitrary:

- `minimum(1, 2, 0)` returns 0
- `minimum(2, 7, 1, 9, 4, -1, 4)` returns -1
- `minimum(0)` returns 0

Exercise 5.8. Write a recursive function for computing the i-th element in a Fibonacci sequence. What happens if a negative value of i is given? What happens if the i is not an integer?

Exercise 5.9. Examine the code used for computing satellite position from broadcast ephemerides or from the almanach. Think how can it be transformed into a function. What should be the arguments? What will be a return value?

Exercise 5.10. Write a function computing the factorial of a natural number. Use either recursive or iterative approach.

Exercise 5.11. Write a piece of code that reads the integer values (user's input) to a six-element list. Then write a function, that:

- checks if the values in a list are sorted in ascending or descending order (returns True/False).
- checks if a list is symmetrical (like a palindrome) (returns True/False).
- checks if there is a repetition of at least one element in the list (returns True/False).

Can the above function be used for a list containing characters (letters/numbers)?

Exercise 5.12. Create a function computing value of a polynomial of the order n for value x. Arguments of the function are coefficients of the polynomial and x.

Exercise 5.13. Write a function `product_all` that takes an arbitrary number of arguments and returns their product.

5.9 Useful links

- [**Recommended**] Local and global scope in examples:
  `https://nbviewer.jupyter.org/github/rasbt/python_reference/blob/master/tutorials/scope_resolution_legb_rule.ipynb`
- Functions:
  - Functions in Python documentation:
    `https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions`
  - Types of arguments:
    `https://levelup.gitconnected.com/5-types-of-arguments-in-python-function-definition-e0e2a2cafd29`
  - `https://www.learnpython.org/en/Functions` (some interactive exercises included)
  - `https://www.w3schools.com/python/python_functions.asp`
  - `https://365datascience.com/tutorials/python-tutorials/python-functions/` (nice examples included)
  - `https://realpython.com/defining-your-own-python-function/`
  - `https://realpython.com/python-return-statement/`
  - `https://www.datacamp.com/community/tutorials/functions-python-tutorial` (examples included)
- More on annotations:
  - `https://www.python.org/dev/peps/pep-3107/`
  - `https://stackoverflow.com/questions/3038033/what-are-good-uses-for-python3s-function-annotations`
- None Type:
  - `https://www.educative.io/edpresso/what-is-the-none-keyword-in-python`
  - https://realpython.com/null-in-python/
- Passing an object to function:
  `https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/`
- Asterisk operator: `https://treyhunner.com/2018/10/asterisks-in-python-what-they-are-and-how-to-use-them/`
- Objects and references in Python:
  - `http://foobarnbaz.com/2012/07/08/understanding-python-variables/`
  - `https://runestone.academy/runestone/books/published/thinkcspy/Lists/ObjectsandReferences.html`
- Docstring:
  - official guidelines: `https://www.python.org/dev/peps/pep-0257/`
  - `https://realpython.com/documenting-python-code/`
  - Python community answer:
    `https://stackoverflow.com/questions/3898572/what-is-the-standard-python-docstring-format`
- `lambda` functions:
  - Video: `https://realpython.com/lessons/what-is-lambda-function/`
  - `https://realpython.com/python-lambda/`
- [**Recommended**] Video lessons on functions: `https://www.youtube.com/watch?v=u-OmVr_fT4s`

References

Beazley, D. M. (2009). *Python essential reference*. Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book*. Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].