Chapter 4: Functions

We have already used (*called*) functions multiple times, like `print()`, `int()`, `len()` or `randint()`.

In [1]:

```
import random

name = "James Bond"
number = int("007")
print(name)
print(len(name))
print(number)
print(random.randint(1, 100))
```

```
James Bond
10
7
67
```

The concept of a function in programming is very close to the mathematical definition of a function. These functions can:

- accept 0, 1 or multiple parameters;
- return a value or not;
- meanwhile causing *side-effects*, like printing a message on the console output.

Defining custom functions

By defining custom functions, the redundancy in the code can be reduced. A custom function can be defined with the `def` keyword:

```
def function_name ( <parameter_list> ):
    function_statement
```

By *defining* a function we are just "storing" it, be we are not executing it yet. For example:

In [2]:

```
def hello():
    print("Hello World!")
```

Now we may *call* the function even multiple types to execute it:

In [3]:

```
print("First line")
hello()
print("Second line")
hello()
```

```
First line
Hello World!
Second line
Hello World!
```

What will be the type of a function?

In [4]:

```
print(type(hello))
```

```
<class 'function'>
```

Parameters

Functions may have zero, one or multiple parameters, which are given between parentheses as *variables* to the function:

In [5]:

```
def greet(name):
    print("Hello " + name + "!")
```

In [6]:

```
greet("John")
someName = "Jane"
greet(someName)
```

```
Hello John!
Hello Jane!
```

In the above example the variable `name` is a **parameter**. The literal value `John` and the variable `someName` are the **arguments** of the function call. So parameters are the generalized variables in the function definitions, while arguments are the actual, concrete values in a function call.

Return values

Functions can return a value with the `return` statement. When a function reaches a return statement, the execution of the function is stopped and the given value is returned. (A function can contain multiple return statements when using conditions or iterations.)

Let's write the `sum_list` function, which receives a list of numerical values as a parameter and returns the sum of the numbers!

In [7]:

```
def sum_list(numbers): # numbers is assumed to be a list of numerical values
    sum_value = 0
    for num in numbers:
        sum_value += num
    return sum_value
    print("This line will never get printed")
```

Until now, we have only defined the function, now we can call it:

In [8]:

```
nums = [12, 8, 37, 21, 67, 42, 25]
print(sum_list(nums))
```

```
212
```

A function can contain multiple `return` statements. After the first `return` statement reached, the execution of the function is stopped.

Let's write the `average` function, which receives a list of numerical values as a parameter and returns the average of the numbers. If the list is empty, the returned value shall be `None`. Reuse the previous `sum_list` function to produce the sum of the values.

In [9]:

```
def average(numbers): # numbers is assumed to be a list of numerical values
    if len(numbers) == 0:
        return None
    else:
        return sum_list(numbers) / len(numbers)
    print("This line will never get printed")
```

In [10]:

```
nums = [12, 8, 37, 21, 67, 42, 25]
print(average(nums))
```

```
30.285714285714285
```

Remark: the `None` keyword is used to define a no value at all (also called *null value*). `None` is not the same as `0`, `False`, or an empty string. `None` has a data type of its own (`NoneType`) and only `None` can be `None`.

Functions returning a value are called *fruitful* functions. Functions without a return value are called *void* functions. In that case the returned value is *None*.

In [11]:

```
greet("Matthew")
result = greet("Andrew")
print(result)
```

```
Hello Matthew!
Hello Andrew!
None
```

Multiple parameters

Functions may have multiple parameters. In such a case the arguments are matched to the parameters in the same order as they are listed.

In [12]:

```
def add(a, b):
    print("Adding {0} and {1}".format(a,b))
    c = a + b
    return c

result = add(10, 32)
print(result)
result = add(-5, 8)
print(result)
```

```
Adding 10 and 32
42
Adding -5 and 8
3
```

Default arguments

Python allows function parameters to have default values. If the function is called without the argument, the parameter gets its default value.

In [13]:

```
def power(base, exp = 10):
    return base ** exp

print(power(2, 6))
print(power(2, 10))
print(power(2))
```

```
64
1024
1024
```

IMPORTANT: if a parameter has a default value, all other parameters following it must have a default value too! E.g. this is **invalid**:

```
def power(base = 2, exp):
    return base ** exp
```

Passing arguments by their position or name

In Python, we can either pass the arguments by their *position* - as we have seen it so far:

In [14]:

```
print(power(2, 6))
print(power(6, 2))
```

```
64
36
```

Alternatively arguments can be passed by the respective *parameter name*:

In [15]:

```
print(power(base = 2, exp = 6))
print(power(exp = 6, base = 2))
print(power(2, exp = 6))
```

```
64
64
64
```

Note: passing arguments by their name is especially useful when:

- a function has many parameters and the function call is much more *readable* when the parameters are passed by their name;
- a function has many parameters with default values and we would like to override the default value for only a few of them.

Built-in functions

There are many built-in functions in Python for common use cases, e.g. for looking up the maximum/minimum value in a list, or to calculate the sum of a list:

In [16]:

```
print("Maximum value in nums: {0}".format(max(nums)))
print("Minimum value in nums: {0}".format(min(nums)))
print("Sum of the values in nums: {0}".format(sum(nums)))
```

```
Maximum value in nums: 67
Minimum value in nums: 8
Sum of the values in nums: 212
```

A comprehensive list can be found in the documentation:
https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

Note: defining a variable or function with the same of an existing (even builtin) function will hide it.

In [17]:

```
print("Maximum value in nums: {0}".format(max(nums)))
max = 42
print("Maximum value in nums: {0}".format(max(nums))) # yields error ,as max in
 an integer now, not a function
```

```
Maximum value in nums: 67

---------------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-17-582e3602d77e> in <module>
      1 print("Maximum value in nums: {0}".format(max(nums)))
      2 max = 42
----> 3 print("Maximum value in nums: {0}".format(max(nums))) # yiel
ds error ,as max in an integer now, not a function

TypeError: 'int' object is not callable
```

Modules

In Python a logical unit of definitions (*variables, functions, classes*) shall be put in a standalone file to support the easy reuse of the code. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module.

There are many built-in modules, we have already used the `math` and the `random` module for example. By using modules we can access preinstalled libraries and use them, so our code will be shorter and more compact.

In [18]:

```
import math
print(math.pi) # using a variable definition from module math
print(math.factorial(10)) # using a function definition from module math
```

```
3.141592653589793
3628800
```

You can easily get a documentation for a module, by either looking it up in the reference: https://docs.python.org/3/library/math.html (https://docs.python.org/3/library/math.html)

Or fetching it dynamically with the `help` function:

In [19]:

```
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

    ...

    sqrt(x, /)
        Return the square root of x.

    tan(x, /)
        Return the tangent of x (measured in radians).

    tanh(x, /)
        Return the hyperbolic tangent of x.

    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.

        Uses the __trunc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    (built-in)
```

Summary exercises on functions

Task 1: Fahrenheit to Celsius

Write a function `fahr2cels`, which computes the temperature in Celcius from Fahrenheit. The formula is the following:

$$C = \frac{5}{9} * (F - 32)$$

Where C is the degree in Celsius and F is the degree in Fahrenheit.

Write a program which prints out the appropriate Celsius values for each degree in Fahrenheit between 0 and 100, using an incremental step of 10.

In [20]:

```
def fahr2cels(f):
    c = 5 / 9 * (f - 32)
    return c

for fahr in range(0, 101, 10):
    cels = fahr2cels(fahr)
    print("Fahr = {0}, Cels = {1:.4f}".format(fahr, cels))
```

```
Fahr = 0, Cels = -17.7778
Fahr = 10, Cels = -12.2222
Fahr = 20, Cels = -6.6667
Fahr = 30, Cels = -1.1111
Fahr = 40, Cels = 4.4444
Fahr = 50, Cels = 10.0000
Fahr = 60, Cels = 15.5556
Fahr = 70, Cels = 21.1111
Fahr = 80, Cels = 26.6667
Fahr = 90, Cels = 32.2222
Fahr = 100, Cels = 37.7778
```

Task 2: Prime check

Write a function `isPrime` which determines whether a number received as a parameter is a prime or not. (You may reuse your algorithm from the previous lecture.)

Wrtite a program which request a number from the user and tests whether it is a prime or not. Check whether the user input is really an integer number or not.

In [21]:

```
import math

def isPrime(number):
    # Handle 0 and 1 as a special case
    if number < 2:
        return False

    # Numbers >= 2 are tested whether they have any divisors
    for i in range(2, int(math.sqrt(number) + 1)):
        #print("Testing divisor %d" % i)
        if number % i == 0:
            # If we found a divisor, we can stop checking, because the number is
 NOT a prime
            return False

    # If no divisors were found, then the number is a prime
    return True

try:
    num = int(input("Number to check: "))
    if isPrime(num):
        print("{0} is a prime".format(num))
    else:
        print("{0} is NOT a prime".format(num))
except:
    print("That was not a number!")
```

```
37 is a prime
```

Task 3: Word count

Request a string input from the user (a sentence). Write a function `wordCount` which count the words in the sentence!

In [22]:

```
def wordCount(sentence):
    spaceCount = 0
    for char in sentence:
        if char == ' ':
            spaceCount += 1
    return spaceCount + 1

userInput = input('Say a sentence: ')
print('Your sentence consisted of {0} words.'.format(wordCount(userInput)))
```

```
Your sentence consisted of 9 words.
```

Hint: count the spaces in the input string.

Task 4: Monotonity

A) Given a list a numbers, write a function `isMonotonous` which decides whether the sequence is monotically increasing or not?

Sample input:

In [23]:

```
list1 = [10, 20, 50, 400, 600]
list2 = [10, 20, 50, 40, 600]
list3 = [1000, 500, 200, 50, 10]
list4 = [10, 20, 50, 50, 300]
```

In [24]:

```
def isMonotonous(numbers):
    # Assume that the list is monotically increasing and search for an index pai
r where it is not true!
    for i in range(1, len(numbers)):
        if numbers[i - 1] > numbers[i]:
            return False
    # If no such errornous index pair was found, then the list was really monoti
cally increasing.
    return True

print("List 1: {0}".format(isMonotonous(list1)))
print("List 2: {0}".format(isMonotonous(list2)))
print("List 3: {0}".format(isMonotonous(list3)))
print("List 4: {0}".format(isMonotonous(list4)))
```

```
List 1: True
List 2: False
List 3: False
List 4: True
```

B) Modify the previous function, so it decides whether the sequence is monotonous or not. (It can be either increasing or decreasing.)

In [25]:

```
def isMonotonous(numbers):
    # Check for monotically increasing
    isIncreasing = True
    for i in range(1, len(numbers)):
        if numbers[i - 1] > numbers[i]:
            isIncreasing = False
            break

    # Check for monotically decreasing
    isDecreasing = True
    for i in range(1, len(numbers)):
        if numbers[i - 1] < numbers[i]:
            isDecreasing = False
            break

    # Return whether either one of the 2 conditions were true!
    return isIncreasing or isDecreasing

print("List 1: {0}".format(isMonotonous(list1)))
print("List 2: {0}".format(isMonotonous(list2)))
print("List 3: {0}".format(isMonotonous(list3)))
print("List 4: {0}".format(isMonotonous(list4)))
```

```
List 1: True
List 2: False
List 3: True
List 4: True
```