Fundamentals of the Python language

Lecture schedule

1. Basic data types

2. Variables

3. Strings

4. Instructions

- Conditional
- Loops
- Selection
- Exception

5. Collections

- Lists
- Dictionary
- Set
- Tuple

6. Functions

7. File handling

8. Object-oriented programming

1. Basic data types

The following basic data types are available in Python:

- int - integers
- float - real numbers
- str - strings
- bool - logical
- complex - complex numbers
- None - *NULL*

A function is used to check the data type: `type()`

In [ ]:

```
print(type(5))
print(type(5.5))
print(type("ABCD"))
print(type(True), type(False))
print(type(1 + 2j))
print(type(None))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'> <class 'bool'>
<class 'complex'>
<class 'NoneType'>
```

Type conversion

In [ ]:

```
a = int("10")
print(a)
print(type(a))
a = int("1010", base=2)
print(a)
print(type(a))
```

```
10
<class 'int'>
10
<class 'int'>
```

In [ ]:

```
b = str(20)
print(b)
print(type(b))
b = bin(20)
print(b)
print(type(b))
b = oct(20)
print(b)
print(type(b))
b = hex(20)
print(b)
print(type(b))
```

```
20
<class 'str'>
0b10100
<class 'str'>
0o24
<class 'str'>
0x14
<class 'str'>
```

In [ ]:

```
c = float("2.6")
print(c)
print(type(c))
c = float(10)
print(c)
print(type(c))
```

```
2.6
<class 'float'>
10.0
<class 'float'>
```

In [ ]:

```
d = str(5.4)
print(d)
print(type(d))
```

```
5.4
<class 'str'>
```

In [ ]:

```
compl = complex("2+3j")
print(compl)
print(type(compl))
```

```
(2+3j)
<class 'complex'>
```

2. Variables

Python does not have a command to declare a variable.

A variable is created when a value is first assigned to it. At the same time, the type of the variable is determined based on the value.

Variable names are case-sensitive..

Using a previously undefined variable raises a `NameError` exception.

A defined global variable is also available inside a function. The `global` keyword allows you to create global variables inside a function.

In [3]:

```
variable = 10
print(variable)
print(type(variable))
variable:int = 10
print(variable)
print(type(variable))
variable = 2.5
print(variable)
print(type(variable))
try:
  value = variable
except NameError as e:
  print(e)
  print("Variable names are case sensitive.")

def function1():
  print("The global variable is available inside the function.")
  print(variable)

function1()

def function2():
  global global_variable
  global_variable = 10000
  print("Calling the function sets a global variable.")

function2()

print(global_variable)
```

```
10
<class 'int'>
10
<class 'int'>
2.5
<class 'float'>
name 'Variable' is not defined
Variable names are case sensitive.
The global variable is available inside the function.
2.5
Calling the function sets a global variable.
10000
```

3. Strings

Strings in Python are enclosed in double quotes or single quotes. Multi-line strings are delimited using triple quotes or single quotes.

In Python, there is no separate type for a single character - it is treated as a string of length 1.

Strings are immutable sequences - you cannot directly change the character(s) in a string.

In [4]:

```python
lan1 = "String in double quotes"
print(lan1)
lan2 = 'String in single quotes'
print(lan2)
lan3 = """Multi-line text,
and another line,
and last one"""
print(lan3)
escape_sequences = "\n Text \tasdasd    "
print(escape_sequences)
```

```
String in double quotes
String in single quotes
Multi-line text,
and another line,
and last one

 Text    asdasd
```

Some actions and functions for strings

Strings cutting

In [5]:

```
lan1 = "String in double quotes"
print(lan1[1:3])
print(lan1[-2:])
print(lan1[:-2])
print(lan1[::2])
```

```
tr
es
String in double quot
Srn ndul uts
```

Strings concatenating

In [7]:

```
lan4 = lan1 + " " + lan2
print(lan4)
table = ["Tekst1", "Tekst2", "Tekst3"]
print(" $ ".join(table))
```

```
String in double quotes String in single quotes
Tekst1 $ Tekst2 $ Tekst3
```

Strings splitting

In [8]:

```
print(lan1.split())
print(lan3.splitlines())
```

```
['String', 'in', 'double', 'quotes']
['Multi-line text,', 'and another line,', 'and last one']
```

Leading and trailing whitespaces removing

In [9]:

```
print(escape_sequences.strip())
print(escape_sequences.rstrip())
print(escape_sequences.lstrip())
```

```
Text    asdasd

 Text   asdasd
Text    asdasd
```

String cases

In [10]:

```
print(lan1.upper())
print(lan1.lower())
print(lan1.title())
print(lan1.capitalize())
```

```
STRING IN DOUBLE QUOTES
string in double quotes
String In Double Quotes
String in double quotes
```

String formatting

In [11]:

```
print("I am {} years old.".format(10))
print("I am {} years and {} months old.".format(10,20))
```

```
I am 10 years old.
I am 10 years and 20 months old.
```

f-String

In [ ]:

```
print(f"I am {10} years and {20} months old.")
var = 3.6354
print(f"Variable value: {var:.3}")
```

```
I am 10 years and 20 months old.
Variable value: 3.64
```

4. Instructions

Typical logical conditions are used in the instructions:

`a < b`, `a <= b`, `a > b`, `a >= b`, `a != b`, `a == b`

and logical operators:

`and`, `or`, `not`, `in`, `is`

Conditional

The conditional instruction `if` in its most elaborate version has the form:

```
if condition1:
  instruction1
  instruction2
  ...
elif condition2:
  instruction12
  instruction22
  ...
elif condition3:
  ...
elif conditionn:
  ...
else:
  instruction_a
  instruction_b
  ...
```

Indentation in Python indicates which instructions refer to which condition. Lack of indentation will cause an `IndentationError` error. At least one instruction (indentation) must occur for each condition, if no instruction is to be executed, use the `pass` keyword. Conditional instructions can be nested.

A one-line `if` statement is also available:

`instruction1 if condition else instruction2` lub `instruction if condition`

which is often used in list expressions.

In [12]:
```python
a = int(input("Input number a: "))
b = int(input("Input number b: "))
```

```
Input number a: 100
Input number b: 200
```

In [13]:
```python
if a > b:
  print("a > b")
else:
  print("a <= b")

if a > b:
  print("a > b")
elif a < b:
  print("a < b")
else:
  print("a == b")
```

```
a <= b
a < b
```

In [14]:

```python
print("a > b") if a > b else print("a <= b")
print("a > b") if a > b else print("a < b") if a < b else print("a == b")
```

```
a <= b
a < b
```

In [15]:

```python
c = int(input("Input number c: "))
```

```
Input number c: 300
```

In [16]:

```python
if a + b > c and b + c > a and c + a > b:
  print(f"A triangle can be formed from the sides a:{a}, b:{b}, c:{c}.")
else:
  print(f"From the sides a:{a}, b:{b}, c:{c} can not be formed a triangle.")
```

```
From the sides a:100, b:200, c:300 can not be formed a triangle.
```

In [17]:

```python
table = [1,2,3,4,5]
if a in table:
  print(f"a:{a} is in the table {table}")
else:
  print(f"a:{a} is not in the table {table}")
```

```
a:100 is not in the table [1, 2, 3, 4, 5]
```

Loops

The Python language provides 2 loop instructions:

- `while`
- `for`

Loop `while`

The form of the `while` loop:

```
while condition:
  instruction1
  instruction2
  ...
  instructionn
else:
  instructions_condition_False
  ...
```

The `while` loop runs as long as the given condition is true (`True`). The `else` clause is optional and is executed when the loop condition takes the value false (`False`).

In [18]:

```
number = 1

while number <= 5:
  print(number**2)
  number += 1
else:
  print('------')

number = 0
while number <= 100:
  number += 1
  if number % 2:
    continue
  print(number)
  if number == 10:
    break
```

```
1
4
9
16
25
------
2
4
6
8
10
```

Loop `for`

The `for` loop in Python is used to iterate sequences such as strings, lists, dictionaries, tuples and collections.

Form of `for` loop:

```
for variable in sequence:
  instruction1
  instruction2
  ....
```

In [19]:
```
for character in "String":
  print(character)

numbers = ['one', 'two', 'three', 'four']

for number in numbers:
  print(number)

for ind, number in enumerate(numbers, start=1):
  print(ind, number)

for character, number in zip(numbers, ['one', 'two', 'three', 'four']):
  print(character + ' ' + number)
```

```
S
t
r
i
n
g
one
two
three
four
1 one
2 two
3 three
4 four
one one
two two
three three
four four
```

Switch

Python, as of version 3.10, supports a switch instruction that has the following form:

```
match parametre:
    case pattern 1:
        instructions
    case pattern 2:
        instructions
    ...
    case _:
        instructions when there is no match
```

In [21]:

```python
name = input("Input name:")

match name:
  case "Nicholas" | "Xavier" | "Timothy":
    print("Hello son")
  case "Evelina":
    print("Hello niece")
  case "Michael":
    print("Hello nephew")
  case _:
    print("Hello stranger")
```

```
Input name:ABCD
Hello stranger
```

Exception handling

Python allows exception handling. To do so, place potentially dangerous code in the `try` - `except` statement.

The exception handling instruction consists of the following blocks:

- Block `try` - instructions that can trigger an exception,
- Block `except` - instructions executed after exception interception (there may be several blocks of this type)

  `except:` - capture all exceptions not yet handled,

  `except exception_name:` - capture an exception with the given name,

  `except (ex1, ex2, ex3):` - handle several exceptions
- Block `else` - instructions executed if no exception occurs

- Block `finally` - instructions executed after exception handling

In [24]:

```
number = int(input("Input number:"))
try:
  a = 1/number
  if number < 0:
    raise Exception("Negative number.")
except ZeroDivisionError as ex:
  print("Cannot divide by 0")
  print("Error: " + ex.args[0])
except Exception as ex:
  print("Error: " + ex.args[0])
except:
  print("Some other exception")
else:
  print("Everything went well")
finally:
  print("Exception handling finished")
```

```
Input number:20
Everything went well
Exception handling finished
```

5. Collections

Python provides the following built-in data collections:

- Lists
- Dictionaries
- Tuples
- Sets

Lists

Lists are used to store multiple elements under a single variable.

Properties of lists:

- Ordered,

- Indexed from 0,
- Changeable,
- May contain duplicates,
- May contain values of different types.

List declaration:

`a = []`

or

`a = list()`

Creating a list object

In [25]:

```
a = [1,2,3]
print(type(a))
print(a)
b = []
print(type(b))
print(b)
b = list()
print(type(b))
print(b)
```

```
<class 'list'>
[1, 2, 3]
<class 'list'>
[]
<class 'list'>
[]
```

Creating copies of the collection

Assigning a variable of a collection object, such as a list, to another variable does not create a copy of the data, but only assigns a reference to a memory location.

Changes made to a collection object using variables storing a reference to it are performed on the original values.

In [27]:

```
a = [1,2,3]
b = a
print(b)
```

```
c = b
b[1] = 5
print(a)
print(c)
c[2] = 6
print(a)
print(b)
```

```
[1, 2, 3]
[1, 5, 3]
[1, 5, 3]
[1, 5, 6]
[1, 5, 6]
```

There are two types of copies of collection objects available in Python:

- shallow copy
- deep copy

Shallow copy

A copy of all elements of the collection is created, whereby if the element is another collection e.g. another list, only the reference to that list is copied.

In [28]:

```
a = [1,[1,2,3],3]
b = list(a)
print(b)
b[0] = 5
print(a)
print(b)
b[1][0] = 5
print(a)
print(b)

a = [1,[1,2,3],3]
b = a.copy()
print(b)
b[0] = 5
print(a)
print(b)
b[1][0] = 5
print(a)
print(b)

a = [1,[1,2,3],3]
b = a[:]
```

```
print(b)
b[0] = 5
print(a)
print(b)
b[1][0] = 5
print(a)
print(b)
```

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

Deep copy

A full copy of all elements is created.

In [29]:

```
from copy import deepcopy
a = [1,[1,2,3],3]
b = deepcopy(a)
print(b)
b[0] = 5
print(a)
print(b)
b[1][0] = 5
print(a)
print(b)
```

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

List operations

Merging lists

In [ ]:

```
a = [4] + a
print(a)
a = a + [1,3]
print(a)
a.extend([20, 30])
print(a)
```

```
[4, 1, [1, 2, 3], 3]
[4, 1, [1, 2, 3], 3, 1, 3]
[4, 1, [1, 2, 3], 3, 1, 3, 20, 30]
```

Adding and inserting elements

In [30]:

```
a.append(20)
print(a)
a.insert(1, 100)
print(a)
```

```
[1, [1, 2, 3], 3, 20]
[1, 100, [1, 2, 3], 3, 20]
```

Deletion of elements

In [31]:

```
print(a)
a.pop(2)
print(a)
a.remove(20)
print(a)
b.clear()
print(b)
```

```
[1, 100, [1, 2, 3], 3, 20]
[1, 100, 3, 20]
[1, 100, 3]
[]
```

Sorting and counting

In [32]:

```
a= [3,1,7,9]
print(a)
a.sort()
print(a)
print(a.count(20))
print(a.count(3))
```

```
[3, 1, 7, 9]
[1, 3, 7, 9]
0
1
```

Selecting items from a list

In [33]:

```
b.clear()
for x in range(1,11,1):
  b.append(x)
print(b)
print(b[1:5])
print(b[:5])
print(b[5:])
print(b[:-2])
print(b[-2:])
print(b[::2])
print(b[1::3])
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 4, 5]
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8]
[9, 10]
[1, 3, 5, 7, 9]
[2, 5, 8]
```

Lists expressions

Python provides so-called list expressions, which allow lists to be created based on other sequence objects in a simplified way.

In [36]:

```
a= [10,20,30,41,51,60]
b.clear()
for i in a:
  if i % 10 == 0:
    b.append(i)

print(b)
b = [i for i in a if i % 10 == 0]
print(b)
```

```
x = [i for i in range(1,11,1) if i % 2]
print(x)
```

```
[10, 20, 30, 60]
[10, 20, 30, 60]
[1, 3, 5, 7, 9]
```

Dictionaries

A dictionary is a collection of key : value pairs.

Properties of dictionaries:

- Ordered (as of Python 3.7),
- Changeable,
- Keys must not contain duplicates,
- May contain values of different types.

Dictionary declaration:

`s = {}`

or

`s = dict()`

Creating a dictionary object

In [37]:

```
s = {}
print(type(s))
s = dict()
print(type(s))
s1 = {'a':10, 'b': 20}
print(s1)
s2 = {1:10, 2:20, '1':30}
print(s2)
```

```
<class 'dict'>
<class 'dict'>
{'a': 10, 'b': 20}
{1: 10, 2: 20, '1': 30}
```

Access to dictionary items

In [38]:

```
print(s1['a'])
s2[3] = -10
print(s2)

print(s1.get('a'))
print(s1.get('d'))
print(s1.get('d', 0))

print(s2.items())
print(s2.keys())
print(s2.values())
print(*s2.items())
print(*s2.keys())
print(*s2.values())

for key, item in s2.items():
  print(f"Key: {key} - value: {item}")
```

```
10
{1: 10, 2: 20, '1': 30, 3: -10}
10
None
0
dict_items([(1, 10), (2, 20), ('1', 30), (3, -10)])
dict_keys([1, 2, '1', 3])
dict_values([10, 20, 30, -10])
(1, 10) (2, 20) ('1', 30) (3, -10)
1 2 1 3
10 20 30 -10
Key: 1 - value: 10
Key: 2 - value: 20
Key: 1 - value: 30
Key: 3 - value: -10
```

Deleting a dictionary item

In [39]:

```
s2 = {1:10, 2:20, '1':30, 3:-10}

del s2[3]
print(s2.pop(2))
print(s2)
```

```
20
{1: 10, '1': 30}
```

Tuples

Tuples are used to store multiple values assigned to a single variable.

Properties of tuples:

- Orderly,
- Indexed from 0,
- Unchangeable,
- May contain duplicates,
- May contain values of different types.

Declaration of tuple:

`s = ()`

or

`s = tuple()`

Important:

When declaring a **single-element tuple**, remember to add a comma after the element:

`t = (1,)`

Creating a tuple object

In [40]:

```
tup = (1, 2, "ABCD", 2.7)
print(tup)

tup_2 = tuple(tup)
print(tup_2)
```

```
(1, 2, 'ABCD', 2.7)
(1, 2, 'ABCD', 2.7)
```

Access to the elements of the tuple

In [41]:

```
# Access by index as for lists
print(tup[1:3])
print(tup[::2])
```

```
(2, 'ABCD')
(1, 'ABCD')
```

Changing the value of tuple elements

In [42]:

```
try:
  tup[0] = 2
except TypeError as ex:
  print(ex)
  print("The tuple does not allow the value to be changed")

# In order to change a value, you need to convert it to a list and then back to a tuple
l = list(tup)
l[0] = 2
tup = tuple(l)
print(tup)
```

```
'tuple' object does not support item assignment
The tuple does not allow the value to be changed
(2, 2, 'ABCD', 2.7)
```

Unpack Tuples

In [43]:

```
print(tup)
(a, b, *c) = tup
print(a,b,c)
(a, *b, c) = tup
print(a,b,c)
(a, *b) = tup
print(a,b)
(*a, b) = tup
print(a,b)
```

```
(2, 2, 'ABCD', 2.7)
2 2 ['ABCD', 2.7]
2 [2, 'ABCD'] 2.7
2 [2, 'ABCD', 2.7]
[2, 2, 'ABCD'] 2.7
```

Sets

Sets are used to store unique values assigned to a single variable.

Properties of sets:

- Unordered,
- Not indexed,
- Unchangeable,
- Must not contain duplicates,
- May contain values of different types.

Declaration of set:

`s = {values}`

or

`s = set()`

Creating a set object

In [44]:

```
set1 = set()
print(set1)

set1 = {"abcd", 1, '1', 1, 2, 3, 3, True, False}
print(set1)

set2 = set(set1)
print(set2)
```

```
set()
{False, 1, 2, 3, '1', 'abcd'}
{False, 1, 2, 3, '1', 'abcd'}
```

Adding elements to a set

In [ ]:

```
set1.add('1')
print(set1)
```

```
{False, 1, 2, 3, 'abcd', '1'}
```

Removing items from a set

In [45]:

```
set1.discard("abcd")
print(set1)
try:
  set1.remove('1')
except KeyError:
  pass
print(set1)
```

```
{False, 1, 2, 3, '1'}
{False, 1, 2, 3}
```

Set operations

In [51]:

```
set1 = {1, 2, 3, 5, 7, 8, 10}
set2 = {1, 2, 4, 6, 8, 9, 11, 12}
```

Union

In [52]:

```
print(set1.union(set2))
print(set1 | set2)
set3 = set(set1)
set3.update(set2)
print("set3_union: " + str(set3))
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
set3_union: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

Intersection

In [53]:

```
print(set1.intersection(set2))
print(set1 & set2)
set3 = set(set1)
set3.intersection_update(set2)
print("set3_inter: " + str(set3))
```

```
{8, 1, 2}
{8, 1, 2}
set3_inter: {8, 1, 2}
```

Difference

In [54]:

```
print(set1.difference(set2))
print(set1 - set2)
set3 = set(set1)
set3.difference_update(set2)
print("set3_diff: " + str(set3))
```

```
{10, 3, 5, 7}
{10, 3, 5, 7}
set3_diff: {3, 5, 7, 10}
```

Symmetric difference

In [55]:

```
print(set1.symmetric_difference(set2))
print(set1 ^ set2)
set3 = set(set1)
set3.symmetric_difference_update(set2)
print("set3_sym_diff: " + str(set3))
```

```
{3, 4, 5, 6, 7, 9, 10, 11, 12}
{3, 4, 5, 6, 7, 9, 10, 11, 12}
set3_sym_diff: {3, 4, 5, 6, 7, 9, 10, 11, 12}
```

6. Functions

Basic information on functions

A function is a program block that is executed when called.

A function can take parameters and return a value.

A function definition in Python consists of a header and instructions placed in a block, which is delimited by code indentation (there are no additional start and end characters in the block).

```
def function_name(parameters):
  instruction_1
  instruction_2
```

```
    ...
    instruction_n
```

If the function is to return a value, use the `return` command, after which the function immediately terminates.

Python allows unnamed (positional) and named (in the form key=value) parameters to be passed. Unnamed parameters are placed before named ones. We can pass any number of unnamed parameters (using in the parameter name *, usually `*args`) and named (using **, usually `**kwargs`).

In [ ]:

```python
def simple_function() -> None:
  print("We have executed the simple function")

a = simple_function()
print(a)
```

```
We have executed the simple function
None
```

In [58]:

```python
def function_2(par1:int, par2:int=10) -> None:
  print(par1 * par2)

function_2(10)
function_2(10, 20)
function_2(par1=30, par2=10)
function_2(40, par2=10)
# function(par1 = 20, 30) # error the parameters named must be after the positional
```

```
100
200
300
400
```

In [ ]:

```python
from typing import Union

def function_3(par:int|float) -> Union[int,float]:
  return par**2

b = function_3(10)
print(b)

def function_4(*args, **kwargs):
  for i in args:
    print(i)
  for key, item in kwargs.items():
    print(str(key) + ":" + str(item))
```

```python
function_4(1,2,3, a=4, b=5, c=6)
function_4(*[1,2,3], **{'i':4, 'j':5, 'k':6})
```

```
100
1
2
3
a:4
b:5
c:6
1
2
3
i:4
j:5
k:6
```

Function Properties

---

1. **Function as parameter**

Python allows a function to be passed as a parameter to another function.

2. **Function, within function**

Python allows you to define a function inside another function.

3. **Function, can return functions**

Python allows functions to be returned from functions.

In [59]:

```python
from typing import Callable

def sum(arg1:int|float, arg2:int|float) -> int|float:
  return arg1 + arg2

def difference(arg1:int|float, arg2:int|float) -> int|float:
  return arg1 - arg2

def product(arg1:int|float, arg2:int|float) -> int|float:
  return arg1 * arg2

def calculate(fun:Callable[[int|float,int|float], int|float],
           arg1:int|float, arg2:int|float) -> int|float:
  return fun(arg1, arg2)

print(calculate(sum, 1, 3))
print(calculate(difference, 5.3,2.6))
print(calculate(product, 10.3,2))
```

```
4
2.6999999999999997
20.6
```

In [60]:

```
def outer() -> Callable[[int|float,int|float], int|float]:
  def inner(arg1:int|float, arg2:int|float)-> int|float:
    return arg1 * arg2
  return inner

new_function = outer()
print(new_function(10, 20))
print(new_function(2.5, 5.6))
```

```
200
14.0
```

Anonymous `lambda` function

The Python language provides an anonymous single-line lambda function.

A lambda function can take any number of arguments, but can only have one expression.

The syntax of the function is as follows:

```
lambda parameters : instruction
```

The `lambda` expression is often used as a parameter for other functions, so-called higher-order functions, e.g. `map`, `filter`, `reduce`, `sorted`, etc.

In [61]:

```
pow = lambda x, y = 2 : x**y
print(pow(10))
print(pow(10,3))
```

```
100
1000
```

Function `map()`

In [62]:

```
li = [1,2,3,4,5]
print(list(map(pow, li)))
print(list(map(lambda x : x**3, li)))

from functools import partial
print(list(map(partial(pow, y=3), li)))
```

```
[1, 4, 9, 16, 25]
[1, 8, 27, 64, 125]
[1, 8, 27, 64, 125]
```

Function `filter()`

In [63]:

```
print(list(filter(lambda x : x % 2 == 0, li)))
```

```
[2, 4]
```

Function `reduce()`

In [64]:

```
from functools import reduce
print(reduce(lambda x, y : x*y, li))
```

```
120
```

Function `sorted()`

In [65]:

```
dictionary = {'a':3, 'c':5, 'b':2, 'd':1, 'e':4}
print(sorted(dictionary))
print(sorted(dictionary.items()))
print(dict(sorted(dictionary.items(), key=lambda x: x[1])))
tuples_list = [(1,2,3), (4,5,6), (3, 1, 4)]
print(sorted(tuples_list, key=lambda x:x[2], reverse=True))
```

```
['a', 'b', 'c', 'd', 'e']
[('a', 3), ('b', 2), ('c', 5), ('d', 1), ('e', 4)]
{'d': 1, 'b': 2, 'a': 3, 'e': 4, 'c': 5}
[(4, 5, 6), (3, 1, 4), (1, 2, 3)]
```

Generator function

Python provides a special type of function, called a generator. A generator allows a function to suspend, resume and return further values based on the stored state. In order to retrieve the next value from the generator, the `next()` function can be used.

A function becomes a generator when we use `yield` instead of using the `return` clause. Using the `return` command terminates the generator (as is the case with normal functions).

It is also possible to send a value to the generator using the `send` function, in which case the `yield` operator is assigned to a variable in the generator.

In [ ]:

```python
def generator(i:int):
  while True:
    yield i*2
    i += 1

g = generator(1)
print(type(g))
print(next(g))
print(next(g))
print(next(g))
g.close()
# print(next(g)) - StopIteration exception, the generator has been terminated
```

```
<class 'generator'>
2
4
6
```

In [66]:

```python
def generator_1(i:int):
  while True:
    if i > 10: return
    yield i*2
    i += 1

for i in generator_1(1):
  print(i)
```

```
2
4
6
8
10
12
14
16
18
20
```

In [67]:

```python
def gen():
    x = 0
    step = 1
    while True:
        y = yield x
        if y is None:
            x = x + step
        else:
            step = y
        if x < 0:
          return

g = gen()

for i in g:
  print(i)
  if i > 9:
    g.send(-1)
```

```
0
1
2
3
4
5
6
7
8
9
10
9
8
7
6
5
4
3
2
1
0
```

Generator expression

Python provides the ability to create generators using a one-line generator expression. To do so, such an expression is placed in parentheses ().

In [ ]:

```
expression = (a for a in range(1,31) if a%3 == 0)
print(type(expression))
for i in expression:
  print(i)
```

```
<class 'generator'>
3
6
9
12
15
18
21
24
27
30
```

Decorator function

Decorators are a very useful tool in Python because they allow you to modify the behaviour of a function or class.

Decorators allow the code of a function to be 'wrapped' by another function, in order to extend its behaviour, without having to permanently modify it.

Applications include:

- logging when a function is called,
- measuring the duration of the function,
- checking whether the user is logged in when the function is called,
- modifying the operation of functions, and others.

In [69]:

```
import functools

def function_decorator(function):
  @functools.wraps(function)
  def inner_function(*arg):
    print(f"\nI call the function: {function.__name__}")
    x = function(*arg)
    print(f"\nThe function {function.__name__} was terminated", end='')
    print(f" with the result {x}") if x is not None else None
    return x
  return inner_function

@function_decorator
def count(x:int) -> int:
  return x*x

@function_decorator
def display(string:str) -> None:
  print(string)

count(10)
display("\nGood day")
```

```
I call the function: count

The function count was terminated with the result 100

I call the function: display

Good day

The function display was terminated
```

In [70]:

```
import functools

def repeat(count:int):
  def funkcja_posrednia(function):
    @functools.wraps(function)
    def inner_function():
      if count > 0:
        for _ in range(count):
          function()
    return inner_function
  return funkcja_posrednia

@function_decorator
@repeat(10)
def display_1() -> None:
  print('-', end='')

display_1()
```

```
I call the function: display_1
----------
The function display_1 was terminated
```

7. File handling

Function `open()`

Python in the standard provides the `open` function, which allows a file to be opened for read and write operations.

In [73]:

```
file = open("file.txt", "w")
text = input("Input text: ")
file.write(text)
file.close()
file = open("file.txt", "r")
print(file.readline())
file.close()
```

```
Input text: ABCD
ABCD
```

Context manager

The file can also be accessed using the context manager, which takes care of releasing resources, e.g. closing the file.

In [ ]:

```
with open("file.txt", "r") as file:
  print(file.read())

with open("file1.txt", "w") as file1, open("file1.txt", "r") as file2:
  file1.write("Example text.")
  file1.flush()
  print(file2.read())
  file2.seek(0)
  print(file2.read())
```

```
ABCDEFGH
Example text.
Example text.
```

Csv and json files

The libraries that come with Python include modules that allow access to many common file formats for storing data, e.g. csv, json, etc.

`import csv`

`import json`

In [75]:

```
import csv
headline = ["c1", "c2", "c3", "c4"]
data = [[1,2,3,4], [5,6,7,8]]

with open('file.csv', 'w') as file:
  writer = csv.writer(file, delimiter=';')
  writer.writerow(headline)
  writer.writerows(data)

with open('file.csv', 'r') as file:
  reader = csv.reader(file, delimiter=';')
  heading_read = next(reader)
  data_read = []
  for row in reader:
    data_read.append(row)
```

```
print(heading_read)
print(data_read)
```

```
['c1', 'c2', 'c3', 'c4']
[['1', '2', '3', '4'], ['5', '6', '7', '8']]
```

In [76]:

```
import json

dictionary = {1:1, 2:2, "sub-dictionary":{"pa":3, "pb":4}}
print(json.dumps(dictionary, indent=4))

with open("file.json", "w") as file:
  json.dump(dictionary, file)

with open("file.json", "r") as file:
  json_object = json.load(file)
  print(json_object)
```

```
{
    "1": 1,
    "2": 2,
    "sub-dictionary": {
        "pa": 3,
        "pb": 4
    }
}
{'1': 1, '2': 2, 'sub-dictionary': {'pa': 3, 'pb': 4}}
```

8. Object-oriented programming

Python supports the possibility of object-oriented programming, although some paradigms are based only on naming conventions.

Class definition

```
class Class_Name(Base_Class):
  class_components
```

In [78]:

```
class Person():
  # class constructor
  def __init__(self, name: str, last_name: str) -> None:
    self.__name = name
    self.__last_name = last_name
  def __str__(self) -> str:
    self.__my_private_function()
    return self.__name + " " + self.__last_name

  @property
  def last_name(self) -> str:
    return self.__last_name

  @property
  def name(self) -> str:
    return self.__name

  @last_name.setter
  def last_name(self, last_name:str):
    self.__last_name = last_name

  @name.setter
  def name(self, name:str):
    self.__name = name

  def __my_private_function(self):
    print("ABCD")

  def _my_protected_function(self):
    print("ABCD")

os = Person("Jan", "Kowalski")
print(os.name, os.last_name)
os.name = "Roch"
print(os.name, os.last_name)
print(os)
```

```
Jan Kowalski
Roch Kowalski
ABCD
Roch Kowalski
```

Inheritance

In [79]:

```
class Employee(Person):
  def __init__(self, name: str, last_name: str, profession: str, age: int) -> None:
    super().__init__(name, last_name)
    self.__profession = profession
    self.__age = age
    self._my_protected_function()
    self._Person__my_private_function()
    print(self._Person__name)

  def __str__(self) -> str:
    return self._Person__name + " " + self._Person__last_name + " " + self.__profession + " " + str(self.__age)

employee = Employee("Grzegorz", "Michalski", "plumber", 40)
print(employee)
```

```
ABCD
ABCD
Grzegorz
Grzegorz Michalski plumber 40
```