You have downloaded a document from
RE-BUŚ
repository of the University of Silesia in Katowice

Title: Python Intermediate

Author: Viera Michaličková, Zenón José Hernández-Figueroa, José Daniel González-Domínguez, Juan Carlos Rodríguez-del-Pino, Jan Přichystal, Kornel Chromiński

Citation style: Michaličková Viera, Hernández-Figueroa Zenón José, González-Domínguez José Daniel, Rodríguez-del-Pino Juan Carlos, Přichystal Jan, Chromiński Kornel. (2021). Python Intermediate. Nitra : Constantine the Philosopher University in Nitra.

Uznanie autorstwa - Użycie niekomercyjne - Bez utworów zależnych Polska - Licencja ta zezwala na rozpowszechnianie, przedstawianie i wykonywanie utworu jedynie w celach niekomercyjnych oraz pod warunkiem zachowania go w oryginalnej postaci (nie tworzenia utworów zależnych).

Python intermediate

Viera Michaličková
Zenón José Hernández-Figueroa
José Daniel González-Domínguez
Juan Carlos Rodríguez-del-Pino
Jan Přichystal
Kornel Chromiński

www.fitped.eu

2021

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Python Intermediate

Published on

November 2021

Authors

Viera Michaličková | Constantine the Philosopher University in Nitra, Slovakia

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Jan Přichystal | Mendel University in Brno, Czech Republic

Kornel Chromiński | University of Silesia in Katowice, Poland

Reviewers

Jozef Kapusta | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

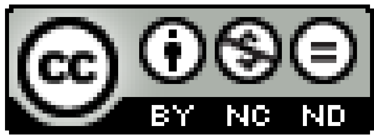
Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



Licence (licence type: Attribution-Non-commercial-No Derivative Works) and may be used by third parties as long as licensing conditions are observed. Any materials published under the terms of a CC Licence are clearly identified as such.

All trademarks and brand names mentioned in this publication and all trademarks and brand names mentioned that may be the intellectual property of third parties are unconditionally subject to the provisions contained within the relevant law governing trademarks and other related signs. The mere mention of a trademark or brand name does not imply that such a trademark or brand name is not protected by the rights of third parties.

© 2021 Constantine the Philosopher University in Nitra

ISBN 978-80-558-1784-2

Table of Contents

1 Exceptions 5
  1.1 Exceptions 6
  1.2 Exceptions (programs) 18
2 Text Files 22
  2.1 Text Files 23
  2.2 Text Files (programs) 33
3 Introduction to Lists 37
  3.1 Introduction to Lists 38
  3.2 Introduction to Lists (programs) 45
4 Storing Data in Lists 48
  4.1 Storing Data in Lists 49
  4.2 Storing Data in Lists (programs) 56
5 Processing Lists (Functions, Methods & Slicing) 61
  5.1 Processing Lists (Functions, Methods & Slicing) 62
  5.2 Processing Lists (programs) 68
6 Lists as Parameters & Return Values 71
  6.1 Lists as Parameters & Return Values 72
  6.2 Lists as Parameters & Return Values I. (programs) 81
  6.3 Lists as Parameters & Return Values II. (programs) 84
7 Functions as Values 91
  7.1 Functions as Values 92
  7.2 Functions as Values (programs) 102
8 Tuples 105
  8.1 Tuples 106
  8.2 Tuples (programs) 117
9 Lists of Lists 121
  9.1 Lists of Lists 122
  9.2 List of Lists (programs) 133
10 Sets 138
  10.1 Sets 139
  10.2 Sets (programs) 146
11 Dictionaries 150
  11.1 Dictionaries 151

11.2 Dictionaries (programs) 162

Exceptions

Chapter 1

1.1 Exceptions

🕮 1.1.1

Let us test the following program, e. g. with the input values 7 and 2:

```
a = int(input('a: '))
b = int(input('b: '))
print(a // b, a % b)
```

The Output:

```
a: 7
b: 2
3 1
```

Correct results (division and remainder) were calculated.

Now again, but with other input values:

The Output:

```
a: 7
b: 0
Traceback (most recent call last):
File ".../test.py", line 3, in <module>
  print(a // b, a % b)
ZeroDivisionError: integer division or modulo by zero
```

An runtime error occurred while executing the program as we tried to divide a number by zero. The program got into an exceptional state (something unexpected or abnormal has happened), so its execution was stopped. We say, that **an exception was raised**. In this case, it was the *ZeroDevisionError* (we can see it from the output message).

In general, when a Python program encounters a situation that it cannot cope with, it raises an exception.

An exception is a Python **object that represents an error**.

📝 1.1.2

There are *many built-in types of errors* in Python.

Match the examples of code with outputs describing a corresponding exception that was raised:

```
>>> 123 + 'x'
_____
>>> 5 / 0
_____
>>> 'hallo'[10]
_____
>>> 'abcdef'.index('x')
_____
>>> 123[0]
_____
>>> fun()
_____
>>> open('data.txt')
_____
>>> def fun(): x += 1
>>> fun()
_____
```

- IndexError: string index out of range
- NameError: name 'fun' is not defined
- UnboundLocalError: local variable 'x' referenced before assignment
- TypeError: unsupported operand type(s) for +: 'int' and 'str'
- ZeroDivisionError: division by zero
- TypeError: 'int' object is not subscriptable
- ValueError: substring not found
- FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'

📖 1.1.3

The **runtime errors** are quite common, and our programs should be able to cope with them.

In our previous simple example, we could check the second value before trying to divide:

```
a = int(input('a: '))
b = int(input('b: '))
if b != 0:
    print(a // b, a % b)
else:
    print('division by zero')
```

But there is a better way of handling such exceptional states in Python using the **try** and **except** blocks.

If we have some suspicious code that may raise an exception, we can defend our program by placing the suspicious code in a *try* block.

The *try* block is followed by an *except* block containing code for handling the problem as elegantly as possible.

When an exception is raised somewhere in the *try* block, all subsequent commands in this block are skipped immediately and the flow of execution proceeds in corresponding *except* block:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)                    # line 1
    print('the results were printed out')   # line 2
except ZeroDivisionError:
    print('division by zero')               # line 3
```

The Output:

```
a: 7
b: 0
division by zero
```

Notice, that after the exception was raised in line 1, the line 2 was skipped and the program's flow was passed to line 3 immediately.

1.1.4

In following program, we are reading numbers from a user repreatedly to calculate and print out their powers.

Complete the code in order to catch *ValueError* properly:

```
while True:
    _____:
        x = int(input('next number: '))
        print(x**2)
    _____ ValueError:
        print('Good bye!')
        break
```

The output for 4 cases of different user inputs:

```
x: 1
1
x: 2
4
x: 10
100
x: end
Good bye!
```

The **ValueError** is raised in case the input cannot be converted to int.

1.1.5

We have handled only the *ZeroDivisionError* so far. But also another exception might be raised:

What about inputting a string that cannot be converted to an int?

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)
except ZeroDivisionError:
    print('division by zero')
```

The input and Output:

```
a: hihi
 Traceback (most recent call last):
 File ".../test.py", line 1, in <module>
   a = int(input('a: '))
 ValueError: invalid literal for int() with base 10: 'hihi'
```

Now, the *ValueError* was raised as the input number was an improper parameter for the *int()* conversion function.

A single *try* statement can have **multiple except statements**. This is useful when the *try* block contains statements that may raise different types of exceptions:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
```

```
    print(a // b, a % b)
except ValueError:
    print('input numbers must be integers')
except ZeroDivisionError:
    print('division by zero')
```

When an exception occurs, it is handled in the corresponding *except* block.

If the exception is of a different type, it is not handled and the program would terminate.

📖 1.1.6

Type of the error can be omitted when specifying the *except* block. In this situation, all kinds of exception would be caught and handled:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)
except:
    print('some exception occured')
```

Or, some specific errors at first, and then catching the other ones:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)
except ValueError:
    print('input numbers must be integers')
except ZeroDivisionError:
    print('division by zero')
except:
    print('some other type of exception was raised ')
```

If we need to handle more types of exceptions in a same way, we can specify them within the same *except* block. But we have to put them in **a tuple**:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)
except (ValueError, ZeroDivisionError):
```

```
    print('input numbers must be integers or division by
zero')
except:
     print('some other type of exception was raised')
```

1.1.7

Study the exception handling logic for lines of code that would be written within the *try* block:

```
try:
    # some code to execute
except Exception1:
    print('A', end='')
except: Exception2:
    print('B', end='')
except (Exception3, Exception4):
    print('C', end='')
except:
    print('D', end='')
print('E')
```

- If an Exception4 had been raised, the output would be CE.
- If an Exception1 had been raised, the output would be A.
- If an Exception5 had been raised, the output would be DE.
- Exceptions of the Exception3 and Exception4 types are always raised together.

1.1.8

Exceptions are objects representing various types of errors.

The object created as a result of an exceptional state (*the exception*) contains information describing the error. We can get access to this object like this:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
    print(a // b, a % b)
except Exception as err:
    print('handling an exception: ', err)    # the err object
can be printed out
```

```
    print(type(err))                              # we can determine
its type
    print(err.args)                               # the additional
information about the cause is packed in a tuple
```

The Output:

```
a: 5
b: 0
handling an exception: integer division or modulo by zero
<class 'ZeroDivisionError'>
('integer division or modulo by zero',)
The err variable is a reference to the actual exception.
```

The *Exception* is a type representing a generic exception from which the other types of exceptions are derived. Later, you will learn how to create your own exceptions.

1.1.9

Compare the following examples:

EXAMPLE 1

```
try:
    x = 5
    y = 'x'
    print(x + y)
except err as TypeError:
    print(err)
```

EXAMPLE 2

```
try:
    x = 5
    y = 'x'
    print(x + y)
except TypeError as err:
     print(err)
```

The Output:

```
unsupported operand type(s) for +: 'int' and 'str'
```

In both of them, an exception of the *TypeError* would be raised. Which of the 2 notations of *except* blocks is correct?

- in EXAMPLE 1
- in EXAMPLE 2

📖 1.1.10

We have already seen, that a single *try* statement can have multiple *except* statements. We can also provide a generic *except* clause, which handles any exception.

Additionaly, after the *except* block, we can include an **else** block. The code in the *else* block executes if the code in the *try* block does not raise an exception. It is a good place for code that does not need the *try* block's protection.

We can also use a **finally** block. The *finally* block is a place to put any code that must execute, whether the *try* block raised an exception or not:

```
try:
    a = int(input('a: '))
    b = int(input('b: '))
except ValueError:
    print('input numbers must be integers')
except ZeroDivisionError:
    print('division by zero')
else:
    print(a // b, a % b)
finally:
    print('this is always printed out')
```

Notice the output when no exception was raised:

```
 a: 10
 b: 2
 5 0
 this is always printed out
```

And the other one, when the *b* is equal to *0*:

```
a: 10
b: 0
division by zero
this is always printed out
```

We will use the *finally* block later, in more complex, real-world programs, typically in order to release the allocated resources (e. g. files or network connections).

1.1.11

Match the blocks concerned with exception handling with correct explanations of their meanings:

try: _____

except: _____

else: _____

finally: _____

- Here goes the code that could cause a problem.
- Here goes the error handling code.
- Here goes the code that is executed if there was no exception.
- Here goes the code that is executed under all circumstances.

1.1.12

Many of the standard functions or methods that you have already used in your programs, may raise some kind of exception.

Let us examine the *index()* method for finding substrings in strings:

```
my_string = 'Python'
input_string = input('> ')
print(my_string.index(input_string))
```

The output for the first input case:

```
> thon
2
```

The output for the second input case:

```
> a
Traceback (most recent call last): File ".../test.py", line 3,
in <module>
print(my_string.index(input_string))
```

```
ValueError: substring not found
```

At first, method returned the index of the substring.

In second case, method raised an exception (*ValueError*), because the substring was not found.

This exception could be handled using the *try* and *except* blocks:

```
try:
    my_string = 'Python'
    input_string = input('> ')
    print(my_string.index(input_string))
except ValueError as err:
    print(err)
```

🕮 1.1.13

When writing our own functions, we should also raise a proper exception when necessary.

Let us write a short function returning a circle's content:

```
import math

def circle_content(radius):
    return math.pi * radius ** 2

print(circle_content(1))       # 3.141592653589793
print(circle_content(-2))      # 12.566370614359172
```

In both cases, a result was returned and printed out. But it is obvious, that the second output is not correct as the radius cannot be a negative number. The function should inform its caller about not being able to return a correct result:

```
import math

def circle_content(radius):
    if radius < 0:
        raise ValueError('The radius of a circle cannot be a 
negative number.')
    return math.pi * radius ** 2
```

Let us call the function now, watch the comments with outputs:

```
try:
    print(circle_content(1))      # 3.141592653589793
    print(circle_content(-2))     # nothing was printed out,
because of the exception
except ValueError as err:
    print(err)                    # The radius of a circle
cannot be a negative number.
```

The **raise** statement is used for raising an exception of a specified type wherever the error is identified.

Usually, we put a descriptive string message inside the round brackets (but this is optional), e. g.:

```
raise ValueError('the input need to be an integer')
raise ZeroDivisionError('fraction’s denominator is zero')
raise TypeError('bad operand')
raise ValueError
```

In the very last example of raising the *ValueError* exception, we did not add a custom message. But the standard message about the problem's cause is still available. We can use the exception's reference within the *except* block.

1.1.14

The *raise* statement can be written more than once within the same *try* block.

- True
- False

1.1.15

The *raise* statement can be used also alone, without specifying any exception type:

```
raise
```

We use this notation to propagate the exception further, to the calling function.

The calling function may

- handle the exception properly (if it contains a suitable *except* block)
- or the exception is passed further.

If "no one cares" about handling the existing exception, the Python intepreter will catch it at least and program will terminate.

In the following example, we try to get an integer number from a user. The user is allowed to make 3 mistakes.

If a *ValueError* occurs, the user is informed by printing a message.

After failing for the third time, we **raise** the *ValueError* exception to the calling function (or to the interpreter in this case). The *print()* command would not be executed for the third time:

```
def read_number():
    n = 0
    while True:
        try:
            return int(input('your number: '))
        except ValueError:
            n += 1
            if n >= 3:
                raise
            print('the input is not an integer!')

read_number()
```

Study the output carefully:

```
your number: a
the input is not an integer!
your number: b
the input is not an integer!
your number: c
Traceback (most recent call last): File "... /test.py", line
12, in <module>
read_number()
File ".../test.py", line 5, in read_number
return int(input('your number: '))
ValueError: invalid literal for int() with base 10: 'c'
```

1.1.16

Study the following program carefully:

```
def fun1(x=None):
    if x is None:
        raise TypeError
    try:
        y = int(x) + 1
        print(y)
    except ValueError:
        print('ValueError')
    print('Goodbye from fun1()')

def fun2():
    try:
        fun1()
    except:
        raise
    print('Goodbye from fun2()')

fun2()
```

Choose correct explanation of the program's execution.

- The fun1() is called without a parameter, so the TypeError is raised. This error is handled by the fun2() function.
- The fun1() is called without a parameter, so the TypeError is raised. But this error is not handled by the fun2() function.
- The ValueError exception might be raised by the int() function in case the TypeError is not raised. Such ValueError is handled in the fun1() function.
- The ValueError exception might be raised by the int() function in case the TypeError is not raised. Such ValueError is not handled in the fun1() function.

1.2 Exceptions (programs)

1.2.1 Calculation of Powers

Write code to calculate the appropriate x^y for two integers x and y (base and exponent).

Use the exception mechanism to handle possible error conditions: if user does not enter an integer, print out the '*ValueError*' string.

Whether or not an exception occurs, always print out the *'bye'* string on the last line of output.

```
Input :4
5
Output:1024
bye
```

```
Input :6
5
Output:7776
bye
```

```
Input :xyz
3
Output:ValueError
bye
```

1.2.2 Greatest Common Divisor

Complete the following program to calculate the largest common divisor of two positive integers.

The program already contains the *gcd ()* function, which implements an algorithm to calculate the largest common divisor. Be careful, the input arguments must be positive integers.

Modify the *gcd ()* function as well as the main program to properly handle possible error conditions:

- user does not enter an integer
- negative number as *gcd ()* input parameter

Use the exception mechanism in your solution.

```
Input :95
10
Output:5
```

```
Input :x123
10
Output:invalid literal for int() with base 10: 'x123'
```

```
Input : -38
```

```
26
Output:arguments must be positive integers
```

start.py

```
# write your code here (finish the solution):

def gcd(a, b):
  """ calculates the greatest common divisor of 2 positive
integers"""
  while a != b:
    if a > b:
      a -= b
    else:
      b -= a
  return a

# main program
x = int(input())
y = int(input())
print(gcd(x, y))
```

1.2.3 Fractions

Create a program that reads 2 integers from the input - numerator and denominator of a fraction.

Compare values of the given numbers and print out the corresponding fraction in the desired form (as shown in examples below).

Use the exception mechanism to handle possible error conditions - respond the same for both, *ValueError* and *ZeroDivisionError* exceptions by printing out the *'error'* string.

```
Input :13
1
Output:13
```

```
Input :20
3
Output:6 2/3
```

```
Input :x123
10
Output:error
```

```
Input :15
15
Output:1
```

```
Input :8
0
Output:error
```

Text Files

Chapter 2

2.1 Text Files

2.1.1

Real world applications process (read and produce) various data that are saved in external files. Many of them are *texts*. The examples of files having a text format include *.txt*, *.csv*, *.html*.

Text file is **a sequence of characters**, including the end of line character '\n'.

Or we can think of a text file as **a sequence of lines**. Then, every line is a sequence of characters that ends with *'\n'*. A text file can contain some empty lines as well.

In general, working with files comprises 3 steps:

1. opening a file
2. reading from or writing to the file
3. closing the file

2.1.2

Select all true statements:

- Text files are sequences of lines.
- Text files are sequences of characters.
- Text files could contain lines with 2 occurances of the '\n' character.
- Text files could contain lines having the zero length.

2.1.3

To open a text file, we call the standard *open()* function:

```
f = open('data.txt', 'r')
```

The *open()* function has usually 2 string arguments:

- name of the file to open and
- the **'r'** in case of opening file **for reading**.

We can specify an absolute or relative path within the first string. But now we have assumed that the input file and the Python script are in the same directory.

The *open()* function returns a reference to the opened file. Or in another words: the **input stream** has been opened and file variable *f* provides necessary connection.

If there is no such file on disk, an exception is raised:

```
try:
    f = open('data.txt','r')
    print(f.read())
except FileNotFoundError as err:
    print(err)
```

For non-existing file, the output would be:

```
[Errno 2] No such file or directory: 'data.txt'
```

In the third, optional argument, we can specify the file‘s encoding, e. g.:

```
f = open('data.txt','r', encoding='utf-8')
```

2.1.4

The open() function will raise an exception. Is it true?

```
f = open('data.txt', 'r', encoding='utf-8')
```

- True
- False
- The question cannot be answered.

2.1.5

First, let us read the following text file

```
Twinkle, twinkle, little star,
how I wonder what you are.
Up above the world so high,
like a diamond in the sky.
```

and print it out on the screen:

```
f = open('poem.txt','r')
print(f.readline())              # 'Twinkle, twinkle, little
star,\n'
```

With *readline()* method, we read the first line of a poem.

This method reads and returns one line. The end of the line character is contained as well. The file object updates actual position in a file after each reading, so the next call would return the second line:

```
print(f.readline())                # 'how I wonder what you
are.\n'
```

After reading all lines, the actual position reaches the end of file.

At the end of a file, *readline()* method returns '' (an empty string).

To read all lines, we often use the *while* statement:

```
f = open('poem.txt','r')
line = f.readline()
while line != '':
    print(line, end='')     # the line contains one '\n'
already
    line = f.readline()
```

Also, this notation is possible:

```
f = open('poem.txt','r')
line = f.readline()
while line:
    print(line, end='')
    line = f.readline()
```

After reading all necessary data, the file **should be closed** immediately:

```
f.close()
```

After closing, all the related resources are released and the file is free to be used by other programs.

2.1.6

What is the output of this program?

```
f = open('input_file.txt','r')
line = f.readline()
while line != '':
    print(line[::-1], end='')
```

```
    line = f.readline()
f.close()
```

- Each line read from the input file was reversed and printed separately on the screen.
- One line was printed on the screen. It contained all characters read from the file, but in reversed order.
- An exception was raised as lines cannot be sliced.

📖 2.1.7

When reading lines from an input file, it can be useful to see its real contents (all whitespace characters including end of lines). The ***repr()*** function is useful for this purpose.

```
f = open('poem.txt','r')
line = f.readline()
while line != '':
    print(repr(line))
    line = f.readline()
f.close()
```

The Output:

```
'Twinkle, twinkle, little star,\n'
'How I wonder what you are.\n'
'Up above the world so high,   \t\n'
'Like a diamond in the sky.\n'
```

Notice, that the third line contains more spaces and a tabulator before the final '*\n*'.

Sometimes, we need to remove all leading and trailing whitespaces before processing a line. We can use the ***strip()*** method for this:

```
f = open('poem.txt','r')
line = f.readline()
while line != '':
    print(repr(line.strip()))
    line = f.readline()
f.close()
```

The Output:

```
'Twinkle, twinkle, little star,'
'How I wonder what you are.'
```

```
'Up above the world so high,'
'Like a diamond in the sky.'
```

Notice, that also the '*n*' characters were removed.

The ***strip()*** method can have an optional parameter – a string with characters to be removed. Also the ***lstrip()*** and ***rstrip()*** variants are available.

2.1.8

Match functions and methods with correct comments explaining some of their features:

open(): _____

close(): _____

strip(): _____

repr(): _____

readline(): _____

- returns a reference to the opened file
- returns a line read from the opened file
- removes all leading and trailing whitespaces from a string
- helps to explore the real contents of a string
- closes connection with the opened file

2.1.9

The *for* loop can be very helpful when processing an input text file.

In case, we know the number of lines, we can repeat the *readline()* call, e. g. 4 times to print all 4 lines of the poem:

```
f = open('poem.txt','r')
for i in range(4):
    print(f.readline(), end='')
f.close()
```

But the *for* loop makes the reading process easier also for cases of unknown number of lines:

```
f = open('poem.txt','r')
for line in f:
    print(line, end='')
f.close()
```

2.1.10

The input file contains some URLs of favourite web pages, one per line, e. g.:

```
www.ukf.sk
www.bbc.com
www.google.sk
```

Complete the following program correctly in order to count all addresses from Slovakia:

```
file = open('urls.txt', 'r')
n = 0
for _____ in _____:
    if line.strip().endswith('.sk'):
        n += 1
print(n)
file.close()
```

2.1.11

Sometimes we need to read the contents of an input file as one string:

```
f = open('poem.txt','r')
text = f.read()
print(len(text))           #  total number of characters
print(text)                #  contents of file
f.close()
```

The *read()* method can have an argument – the number of characters to read:

```
f = open('poem.txt','r')
print(f.read(1))            # first character
print(f.read(5))            # next 5 characters
print(f.read())             # the rest of the file’s contents
f.close()
```

2.1.12

The input file *urls.txt* contains 3 lines:

```
www.ukf.sk
www.bbc.com
www.google.sk
```

The program to analyse:

```
file = open('urls.txt', 'r')
print(file.read(3), end='')          # line 1
print(file.read(), end='')           # line 2
print(file.readline(), end='')       # line 3
file.close()
```

Match numbers of lines containing print statements with correct outputs.

line 1: ____

line 2: ____

line 3: ____

- .ukf.sk\nwww.bbc.com\nwww.google.sk\n
- ''
- www.google.sk
- www

2.1.13

For **writing** characters to a text file, writing mode must by specified using the ***'w'*** string in the ***open()*** function's second argument.

If the file does not exist, it is ***created as empty.***

If the file already exists, its contents is going ***to be overwritten***:

```
f = open('output.txt','w')
f.write('We are learning\n')
f.write('about using text files\n')
f.write('in Python\n')
f.close()
```

When using the ***write()*** method, all end of lines have to be written explicitly.

The three lines of text could be output to a file also by calling the ***write()*** method once:

```
f.write('We are learning\nabout using text files\ninPython\n')
f.close()
```

Be careful and ***always close the opened files***. In case of not closing, you cannot be sure, whether the output is really saved on a disk.

2.1.14

Choose correct explanation of the program's output for input value *n == 5*:

```
import random
fw = open('random_numbers.txt', 'w')
n = int(input('n = '))
for i in range(n):
    fw.write(str(random.randint(1, 9)))
fw.close()
```

- The output file file contains 5 random numbers in one line.
- There are 5 lines in the output file, each of them contains a random number.
- The str() function cannot be called like this.

2.1.15

For writing to a text file, the standard *print()* function is applicable as well.

We can redirect the output from **default output stream** (usually the screen) to an external file using an optional argument:

```
f = open('primes.txt','w')
for x in 2, 3, 5, 7, 11, 13, 17, 19:
    print(f'{x}', end=' ', file = f)
f.close()
```

The output was written only to the *f* file now. The *primes.txt* contains 1 line:

```
2 3 5 7 11 13 17 19
```

2.1.16

Choose correct explanation of the following program's Output:

```
fr = open('input.txt', 'r')
fw = open('output.txt', 'w')

for line in fr:
    if line != '':
        print(line, end='', file=fw)
    print(line, end='')

fr.close()
fw.close()
```

- The output file contains all non-empty lines from input.txt. All lines read from the input file were printed out on the screen.
- The output file contains all non-empty lines from input.txt. All these non-empty lines were printed out on the screen as well.
- All lines from the input file were copied to the output file as well as printed on the screen.

2.1.17

When working with files, the **with** construction is recommended to use:

```
with open('poem.txt','r', encoding='utf-8') as f:
    for line in f:
        print(line, end='')
```

The opened file is referenced with variable *f*. After executing the *with* block, the **file is closed automatically**.

Let us copy the contents of a file. The first file has to be opened for reading, the second one for writing:

```
with open('original.txt', 'r') as fr:
    with open('copy.txt', 'w') as fw:
        fw.write(fr.read())
```

We could enumerate the files to be used also in one line:

```
with open('original.txt', 'r') as fr, open('copy.txt', 'w') as
fw:
```

```
    fw.write(fr.read())
```

2.1.18

Study the following lines of code:

```
with open('input.txt', 'r') as fr, open('output.txt', 'w') as
fw:
    for line in fr:
        if line != '':
            print(line[::-1], end='', file=fw)
        print(line[::-1], end='')
    fw.close()
```

Choose all true statements:

- We forgot to close() the fr file.
- The fw.close() statement is redundant and should be deleted.
- The program copies all non-empty lines from input file to the output file, reversing them first.
- The program copies all empty lines from input file to the output file, reversing them first.

2.1.19

Besides the reading and writing modes, also **appending** to a file is possible.

Let us assume, that there are already 3 lines saved in *names.txt* file:

```
John
Paul
George
```

We will open this file in the appending mode (notice, that the second argument is '**a**'):

```
file = open('names.txt', 'a')
file.write('Ringo\n')
file.close()
```

The actual position in file was set to the end of file immediately after opening it. Then, the fourth line was appended, and the file was closed.

The resulting contents of the *names.txt*:

```
John
Paul
George
Ringo
```

2.1.20

Complete the program correctly to count total number of lines in a file and add this information to its end:

```
with open('names.txt', 'r') as f:
    text = f.read()

print(repr(text))    # 'John\nPaul\nRingo\nGeorge\n'

with open('names.txt', _____) as f:
    n = text.count('\n')
    f.write(str(n))
```

2.2 Text Files (programs)

2.2.1 Counting Vowels

Write a program to count all vowels in an input text file. For non-existing files, handle the *FileNotFoundError* exception correctly.

```
Input : textfile1.txt
Output: 21
```

```
Input : text2.txt
Output: file not found
```

```
textfile1.txt:
This is a test file, contents is not important,
just watch vowels to count.
```

2.2.2 Paying for Text Messages

Write program that calculate the total price of sending file's contents as a series of SMS messages.

A text message (SMS) can contain maximum of 160 characters. There is a text saved in an input file. One SMS costs 0.10 Eur.

Be careful and always print out 2 decimals.

```
Input : textfile1.txt
Output: 0.10
```

```
Input : textfile2.txt
Output: 0.20
```

```
textfile1.txt:
Hallo mum, i am driving home.
```

```
textfile2.txt:
This is a longer message, it has more than 160 characters. So
we have to pay more :-)
This is a longer message, it has more than 160 characters. So
we have to pay more :-)
This is a longer message, it has more than 160 characters. So
we have to pay more :-)
```

⌨ 2.2.3 Longest and Shortest Lines

Write program that determine the numbers of the longest as well as the shortest line in a text file.

Be careful about the Output: not the length of a line, but the ordinal number defining its position in file is needed.

```
Input : textfile1.txt
Output: 3 2
```

```
Input : textfile2.txt
Output: 1 1
```

```
textfile1.txt:
The first line is not very short.
The second line is short.
The third line. It is very very very interesting.
```

```
textfile2.txt:
This is a line.
This is a line.
This is a line.
This is a line.
```

2.2.4 Filtering Empty Lines

Write program to copy the contents of an input text file to the output, but without empty lines.

```
Input : textfile1.txt
Output:
a
b
c
```

```
Input : textfile2.txt
Output:
xyz
```

```
textfile1.txt:
a

b
c
```

```
textfile2.txt:



xyz
```

2.2.5 Happy Birthday

Write program to count persons born in a given month.

Input text file contains a family members - dates of births (one date per line).

```
Input : textfile1.txt 5
Output: 1
```

```
Input : textfile2.txt 2
Output: 2
```

```
textfile1.txt:
1.1.1978
31.10.2008
7.5.2010
22.8.1980
```

```
textfile2.txt:
4.2.1920
13.2.1922
18.3.1926
17.9.1944
12.1.1944
```

2.2.6 Strong Passwords

Write program to find out the number of *strong* passwords in a file.

The input text file contains passwords of many users (one per line).

A *strong* password is longer than 6 characters, contains at least 1 digit, 1 uppercase and 1 lowercase letter.

```
Input : textfile1.txt
Output: 2
```

```
Input : textfile2.txt
Output: 0
```

```
textfile1.txt:
Minmax33
HopkyLupkySupky
CupiLupi7
kukuk
```

```
textfile2.txt:
puK12
77hohohohohoho
patmat007
```

Introduction to Lists

Chapter 3

3.1 Introduction to Lists

3.1.1

List is a standard data structure designed for saving and processing sequences of items.

To create a new list, put comma-separated items between square brackets:

```
temperatures = [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
students = ['Augusta Ada', 'Alan Turing', Edsger W. Dijkstra',
'Donald Knuth', 'Niklaus Wirth', 'Ole-Johan Dahl', 'Kristen
Nygaard']
years = [1945, 1969, 1971, 1978, 1980, 1984, 2012, 2015, 2019]
incoming_person = ['Susan', 'Slovakia', 2019, 52.18, true]
x = 17
y = 9
z = 1989
values = [x, y, z]
```

As seen above, the items in a list can be of same as well as of different types.

To create an empty list, two statements can be used:

```
a = []
```

or

```
a = list()
```

For lists, *type()* function returns the corresponding class name (lists are objects of the *list* type):

```
>>> type(a)
<class 'list'>
```

3.1.2

Choose all the list variables from the options listed.

- a = [True]
- b = [1, 2, 3, 4, 5, 6, 7]
- c = ['spring', 'summer', 'autumn', 'winter']
- d = list()
- e = [0, 10, 'X', 120.5]

- f = []

📖 3.1.3

To access elements in lists, indices are used. Similar to string indices, **list indices start at 0**:

```
a = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday' , 'Sunday']
print(a[0])
print(a[2])
print(a[-1])
print(a[-2])
print(a[len(a)-1])
```

When the above code is executed, it produces the following result:

```
Monday
Wednesday
Sunday
Saturday
Sunday
```

📝 3.1.4

Choose statements that are not applicable for accessing the *'red'* element:

```
colors = ['red', 'green', 'blue', 'white', 'black']
```

- colors[len(colors)-1]
- colors[1]
- colors[0]
- ['red', 'green', 'blue', 'white', 'black'][0]

📖 3.1.5

Two lists can be concatenated using the **+ operator**.

By **joining two lists**, we create a new, longer one with all the first list's elements followed by all the second list's elements, preserving the original order.

Watch the outputs of following console experiments:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = [6, 7, 8]
>>> a + b
[1, 2, 3, 4, 5, 6, 6, 7, 8]
>>> students = ['John', 'Martin']
>>> students =  ['Joseph'] + students
>>> students
['Joseph', 'John', 'Martin']
>>> students + []
['Joseph', 'John', 'Martin']
```

3.1.6

Execute the following lines of code:

```
list1 = [1] + [2, 3] + [4] + [5] + []
list2 = [1]+ [5] + [2, 3, 4]
print(len(list1) == len(list2))
```

Fill in the printed value:

3.1.7

We use the * operator when concatenating the **same list repeatedly**.

Let us compare the outputs:

```
>>> languages = ['C', 'Java', 'Python'] + ['C', 'Java',
'Python'] + ['C', 'Java', 'Python']
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java',
'Python']
>>> languages = ['C', 'Java', 'Python'] * 3
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java',
'Python']
```

Using the * operator makes list initialization easier. E. g. instead of writing 10 times the same zero value, we can use the following notation:

```
>>> counters = [0] * 10
>>> counters
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

3.1.8

Execute the following lines of code:

```
greetings =  4 * ['Hallo']
greetings = greetings + ['Bye'] * 3
print(greetings[len(greetings) // 2])
```

Write down the correct Output:

3.1.9

For testing whether **a list contains a specific item**, we use the **in** operator:

```
>>> languages = ['C', 'Java', 'Python']
>>> 'Pascal' in languages
False
>>> 'Python' in languages
True
>>> 'Java' not in languages
False
```

For strings, we use the **in** operator to test whether a string contains a specific substring:

```
>>> message = 'We can do it'
>>> 'can' in message
True
```

3.1.10

Execute the following lines of code:

```
words = ['We', 'can', 'do', 'it']
if ['can','it'] in words:
    print('yes')
else:
    print('no')
```

Choose all the correct statements:

- The program’s output is no.
- The words variable is a list that contains just strings.
- The in operator cannot be used in conditions.
- The program’s output is yes.

3.1.11

We already know that using the **+** operator combines multiple lists into one.

When processing data in programs, we will often solve the problem of filling the list with values entered from input.

Let us first consider the individual reading of values, e.g. input strings.

First we prepare an empty list:

```
a = []
```

Next, some user enters a string, e. g. *'spring'*:

```
user_input = input()
```

We add a new element to the original list (we join the original list and the one-element list containing the new element):

```
a = a + [user_input]
print(a)                  # Output: ['spring']
```

The user inputs a next element, e. g. a string *'summer'*:

```
user_input = input()
a += [user_input]
print(a)                  # Output: ['spring', 'summer']
```

Now, we used a shorter notation for adding a new element to the end of an existing list using the **+=** operator.

3.1.12

Complete the following program.

First, some user enters a number of elements to be stored in a list. Subsequently, he enters individual elements (integers), one by one. In the last line, contents of the *numbers* list is printed out.

```
n = int(input())
numbers _____ _____
for i in range(n):
    numbers_____ [int(_____)]
print(numbers)
```

Input :

```
4
10
20
30
40
```

Output:

```
[10, 20, 30, 40]
```

- +
- input()
- []
- =
- +=

📖 3.1.13

In practical tasks in this course, we will often need to proccess a sequence of input values that are provided as a single string in which the individual elements are separated by a specific delimiter (spaces, commas etc.).

Consider an input string:

```
10,20,30,40
```

We read it from user and then use the *split ()* string method to get a list of string's parts representing the input values:

```
a = input().split(',')
print(a)                    # Output: ['10','20','30','40']
```

We can see from the listing above that we have obtained a list of strings from the *split()* method. If we need to store them as integers, we must cast all elements to *the int* type. A new list of integers will be constructed easily:

```
user_input = input().split(',')
```

```
a = []
for x in user_Input :
    a += [int(x)]
print(a)                    # Output: [10, 20, 30, 40]
```

If we allow users to enter an empty string, we must ensure, that the list that is being created remains empty as well:

```
user_input = input().split(',')  # for an empty string, we get
the [''] list
a = []
if user_input!= ['']
    for x in user_Input :
        a += [int(x)]
print(a)                         # Output: []
```

If we tried to cast the list element [''] into an integer, it would throw an exception. Therefore, we first checked whether this was the case.

Later we will learn a more practical way of storing values of different types into a list using the mapping function.

3.1.14

Complete the following program correctly.

First, the user enters a string containing decimal numbers separated by semicolons. We want to parse this string and save all decimal numbers to a list. If the input string is empty, we need to create an empty list.

```
data = _____
user_input = input._____
if user_input != _____
    for x in user_Input :
        data _____ [float(x)]
print(data)
```

Input :

```
3.14;12.567;0.77;-23.202
```

Output:

```
[3.14, 12.567, 0.77, -23.202]
```

- +=
- []
- ['']
- split()
- split(';')

3.2 Introduction to Lists (programs)

3.2.1 Sum First and Last Element

Given a list of integers, calculate the sum of its first and last elements.

For empty lists, handle the *IndexError* exception correctly.

```
Input : -1,2,3,4,5
Output: 4
```

```
Input :
Output: list index out of range
```

3.2.2 Neighbours of an Element

For a non-empty list of integers and a number representing an element's index, print *yes* if the selected element is larger than the sum of its two adjacent elements. Otherwise, print *no*.

Be careful about the elements, which do not have two neighbours.

```
Input : 1,2,3,0,7
2
Output: yes
```

```
Input : 1,2,3,0,7
3
Output: no
```

```
Input : 1,2,3,0,7
0
Output: not enough neighbours
```

3.2.3 Concatenating

For two lists of integers, create and print a new list of integers that will start and end with all second's list elements having the contents of the first list 3 times in the middle.

```
innput: 1,2
3,4,5
Output: [1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 1, 2]
```

```
Input :
0,0
Output: [0, 0, 0, 0, 0, 0]
```

start.py

```
# do not modify this part of code:
input_list = input().split(',')
a = []
if input_list != ['']:
  for x in input_list:
    a += [int(x)]

input_list = input().split(',')
b = []
if input_list != ['']:
  for x in input_list:
    b += [int(x)]

# write your code here:
```

3.2.4 Names in a List

There is a list of names called *friends* initialized at the beginning of a program.

Read 2 names from a user. Print *True* (boolean value), if either of the input strings is a member of the list, *False* otherwise.

```
Input : John Robert
Output: True
```

```
Input : Sophia Nina
Output: False
```

```
Input : George Martin
Output: True
```

3.2.5 Length of a Password

User will input a non-empty list of passwords (strings).

Print out the length of a password that is saved just in the middle of the given list.

```
Input : abc123,puk,Valibuk+Miesizelezo7!
Output: 3
```

```
Input : puf777,muf1,Muf89,Pascal_2013
Output: 5
```

Storing Data in Lists

Chapter 4

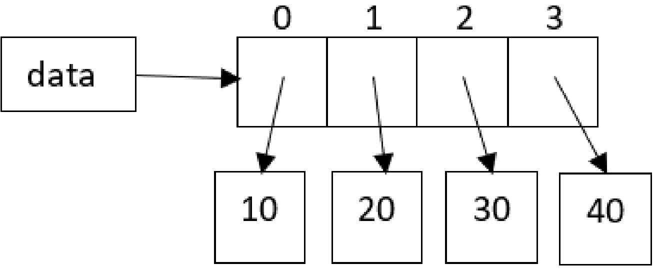
4.1 Storing Data in Lists

4.1.1

Let us consider the following list:

```
data = [10, 20, 30, 40]
```

With this statement, a variable called *data* has been created in global namespace:



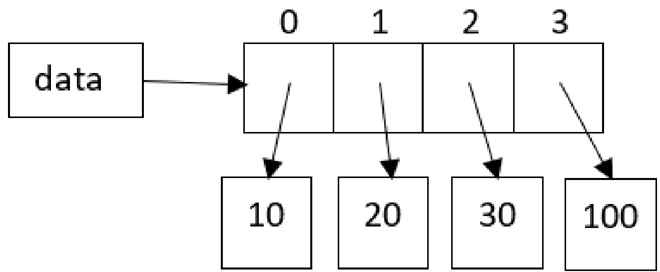
The *data* variable is of a list type. The list itself consists of 4 variables *data[0]*, *data[1]*, *data[2]*, *data[3]*, that reference the four *int* values.

Lists are **mutable structures**. This means, that lists can be modified as necessary. By writing the

```
data[3] = 100
```

statement, we have changed the last element and assigned the corresponding variable a new value.

From now, the variable *data[3]* is referencing the *int* value 100:



4.1.2

Execute the following program:

```
a = [2, 0, 0, 8]
a[3] = 9
a[2] += 1
a = a * 2
print(a)
```

and choose correct output.

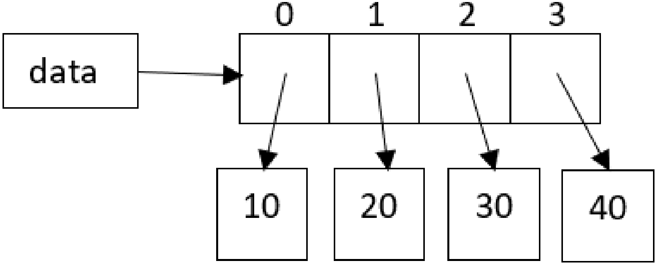
- [2, 0, 1, 9, 2, 0, 1, 9]
- [4,0,0,16]
- [4,0,2,18]
- [[2, 0, 1, 9], [2, 0, 1, 9]]

📖 4.1.3

For deleting a specific element from a list, the *del* statement can be used:

Let us consider the following list:

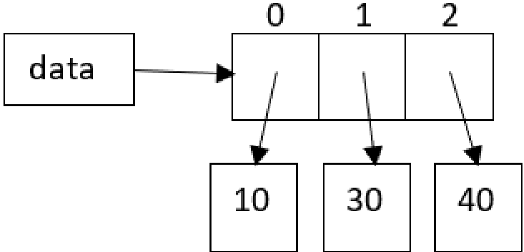
```
data = [10, 20, 30, 40]
```



After executing the following deletion

```
del data[1]
```

the list will contain only 3 elements:



4.1.4

Execute the following lines of code:

```
import random

b = [random.randrange(1,10)] * 10

for i in range(5):
    del b[i]

print(b)
```

Select all correct statements:

- All elements left in the b list are of a same value.
- The program outputs a list with 10 random values.
- The program outputs a list with 5 random values.
- The del statement cannot be applied in this way.

4.1.5

To iterate through a list of elements, we typically use the ***for*** statement. The *i* represents an ***index*** variable. The ***len()*** function returns number of elements in a list.

Let us consider a list of friends' names and print out its content:

```
friends = ['Peter','Paul', 'Michael', 'George', 'John']
for i in range(len(friends)):
    print(f'{i}. {friends[i]}')
```

The output is:

```
0. Peter
1. Paul
2. Michael
3. George
4. John
```

When we are interested only in elements and the indices are not relevant for actual processing, a shorter notation of iteration might be useful:

```
for element in friends:
    print(element, end=' ')
```

In this case, we also replaced the newline characters between elements with spaces, so the output is:

```
Peter Paul Michael George John
```

4.1.6

The *t* list contains a sequence of temperature values.

```
t = [21, 22, 25, 25, 24, 21, 20]
```

We need to calculate the average temperature. Compare the following 2 solutions.

SOLUTION 1

```
sum = 0.0
for i in range(len(t)):
    sum += t[i]
print(f'average temperature is {sum/len(t):.2f}')
```

SOLUTION 2

```
sum = 0.0
for x in t:
   sum += x
print(f'average temperature is {sum/len(t):.2f}')
```

Do they both calculate the correct result?

- True
- False

4.1.7

We want to increment the elements of *t* list by one.

```
t = [21, 22, 25, 25, 24, 21, 20]
```

Consider the following solutions:

SOLUTION 1

```
for x in t:
    x += 1
```

```
print(t)
```

SOLUTION 2

```
for i in range(len(t)):
    t[i] += 1
print(t)
```

Do they both calculate the correct result?

- False
- True

📖 4.1.8

Sometimes we could make use of the *enumerate()* function to access both, the index as well as the corresponding value at one time:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']
for i, element in enumerate(friends):
   print(f'{i}. {element}')
```

The output is:

```
0. Peter
1. Paul
2. Michael
3. George
4. John
```

The *enumerate()* function sets the element counter to *0*, or we can specify the initial counter value using the *start* parameter:

```
for i, element in enumerate(friends, start = 1):
   print(f'{i}. {element}')
```

The output is:

```
1. Peter
2. Paul
3. Michael
4. George
5. John
```

4.1.9

Write down the output of the following program:

```
for i, element in enumerate(['a', 'b', 'c'], start = 1):
    print(i * element, end = ',')
```

Be careful and do not forget to add the comma character after printing each of the calculated values!

4.1.10

List comprehension provides a concise way to create new lists.

It consists of brackets containing an expression followed by a *for* clause, then zero or more *for* or *if* clauses, e. g.:

```
[expression for element in list if condition]
[expression for element in list]
```

With this notation, we get **a new list resulting from evaluating the expression** in context of the *for* and *if* clauses which follow it.

Here are some examples, whatch carefully the outputs:

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [i ** 2 for i in range(5)]
[0, 1, 4, 9, 16]
```

First, a list [0] + [1] + [2] + ... + [9] was created.

Then, the expression *i ** 2* was evaluated for i = 0, 1, 2, 3, 4. So the *int* values 2^0, 2^1, 2^2, 2^3, 2^4 are in the resulting list.

4.1.11

List comprehensions can contain complex expressions and nested functions.

Here are some more examples of using list comprehension, watch the outputs carefully:

```
import random
```

```
a = [random.randint(0, 9) for i in range(10)]
print(a)
```

Output: [4, 4, 9, 7, 4, 1, 0, 0, 3, 6]

```
words = ['this', 'is', 'a', 'list', 'of', 'words']
b = [word[0].upper() for word in words]
print(b)
```

Output: ['T', 'I', 'A', 'L', 'O', 'W']

```
things = ['pen', 'pencil', 'rubber', 'paper']
c = [thing for thing in things if len(thing) > 5]
print(c)
```

Output: ['pencil', 'rubber']

```
from math import pi
d = [str(round(pi, i)) for i in range(1, 6)]
print(d)
```

Output: ['3.1', '3.14', '3.142', '3.1416', '3.14159']

4.1.12

Match the list comprehension notations with their resulting lists (referenced by the *a* variable):

```
a = [0 for i in range(10)]
```

a == ____

```
# the input file contains just lines longer than 30 characters
lines = open('input.txt', 'r').readlines()
a = [line for line in lines if 0 < len(line) < 10]
```

a == ____

```
a = [x * y for x in range(1, 4) for y in range(1, 4)]
```

a == ____

```
years = [1945, 1969, 1971, 1980, 1984, 2012, 2015]
a = [y % 100 for y in years]
```

a == _____

- [1, 2, 3, 2, 4, 6, 3, 6, 9]
- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- []
- [45, 69, 71, 80, 84, 12, 15]

4.2 Storing Data in Lists (programs)

4.2.1 Houses on a Street

There are houses standing on a straight street, one next to another. A list contains numbers of theirs floors.

Find out if there is a house with a specified number of floors. Print out its position or that there is no such house on the street.

```
Input : 1,1,3,2,1,0,2
3
Output: 2
```

```
Input : 1,1,3,2,1,0,2
5
Output: there is no such house
```

4.2.2 Numbers Divisible by a Given Number

For a given list of positive integers and a positive integer *d*, count all the elements divisible by *d*.

```
Input : 3,8,14,33,77,2
7
Output: 2
```

```
Input : 13,22,145
10
Output: 0
```

4.2.3 Average Exam Grade

Students have taken an exam from programming. Their grades are saved in a list of integers. Calculate the average grade.

Be careful about processing the empty list correctly as well as always printing 2 decimals.

```
Input : 1,3,2,5,4
Output: 3.00
```

```
Input : 1,2
Output: 1.50
```

```
Input :
Output: division by zero
```

4.2.4 Increasing Sequence

If the input list of integers represents a *non-decreasing sequence*, print a boolean value *True*, *False* otherwise. The input list is not empty.

```
Input : 1,2,2,5,8,12
Output: True
```

```
Input : -1,0,2,3,0,44,99,50
Output: False
```

```
Input : 5
Output: True
```

4.2.5 Basketball Players

A non-empty input list contains heights of basketball players standing in a row. Find the smallest player. Print out the minimum value as well as its index.

```
Input : 175,175,205,175,167
Output: 167 4
```

```
Input : 205,178,178,201
Output: 178 1
```

```
Input : 185
Output: 185 0
```

4.2.6 Longest Name

A non-empty input list contains names of classmates. Find the one having the longest name. Print out the maximum length, not the longest name.

```
Input : John,Paul,Christopher,Peter
Output: 11
```

```
Input : Max,Kate,Fred
Output: 4
```

```
Input : Steven
Output: 6
```

4.2.7 Shifting

Input list contains integers (0s and 1s). For an integer number *k*, move each element in the list *k* times to the right.

Consider the input list to be a logical circle: When determining new places for elements, end of the list can be achieved soon. In this case, continue position counting from the list's beginning.

Print out the list's status after modifying it.

```
Input : 1,1,0,0,1,0,1
2
Output: [0, 1, 1, 1, 0, 0, 1]
```

```
Input : 0,1,1,0
3
Output: [1, 1, 0, 0]
```

```
Input :
8
Output: []
```

4.2.8 Dice Throw

An experiment was realized. We were throwing one dice repeatedly and results of these throws were saved in a list.

Find out which result had the maximum frequency.

```
Input : 4,2,3,1,4,4,4,4,6,1,5,5
Output: 4
```

```
Input : 1,2,2,6,3,1,6
Output: 1
```

```
Input : 3
Output: 3
```

4.2.9 Arithmetic Sequence

Arithmetic sequence is a sequence of numbers such that the difference between the consecutive members is constant. Difference here means the second minus the first. For instance, the sequence 5, 7, 9, 11, 13, 15, . . . is an arithmetic sequence with common difference of 2.

There are 3 integers given as input values: the first member, the upper bound and the common difference (always a positive number).

Print out a list containing all members of the arithmetic sequence that is defined by these numbers.

```
Input : 5
15
2
Output: [5,7,9,11,13,15]
```

```
Input : -3
10
5
Output: [-3,2,7]
```

4.2.10 Filter of Names

Input list contains names of many friends. Print out a list of those names, that starts with letter '*M*' and are shorter than a given number.

```
Input : John,Paul,Martin,Peter,Max,Mia
4
Output: ['Max','Mia']
```

```
Input : George,Archie,Charlotte,Louis
10
Output: []
```

```
Input : Martin,Maria,Anna
6
Output: ['Maria']
```

4.2.11 Numbers from Interval

Input list contains integers. Print out a list containg all the original list-s elements that are from a given closed interval.

```
Input : 1,4,7,8,3,10,12,13,14,15
4
10
Output: [4,7,8,10]
```

```
Input : 0,0,1,18,2,-2
-5
5
Output: [0,0,1,2,-2]
```

4.2.12 Signs and Zeros

Input list contains integers. Change their signs to the opposite and remove zeros from the list.

Print out the list's status after modifying it.

```
Input : 1,-1,0,7,-4
Output: [-1,1,-7,4]
```

```
Input : 0,0,1
Output: [-1]
```

Processing Lists (Functions, Methods & Slicing)

Chapter 5

5.1 Processing Lists (Functions, Methods & Slicing)

5.1.1

There are some ***standard functions*** in *Python* that might be very useful when processing lists:

- ***len(list)*** returns the number of elements (its length)
- ***sum(list)*** returns the sum of the elements
- ***max(list)*** returns the element with maximum value
- ***min (list)*** returns the element with minimum value

Of course, in some cases, these functions are not applicable (e. g. summing a list of other than number elements or searching for maximum in a list with values of different types) and calling them would lead to an error.

5.1.2

Fill the calls of appropriate functions to calculate correct results:

We need to calculate the average temperature as well as the minimum and maximum values. The *t* list contains a sequence of temperature values:

```
t = [21, 22, 25, 25, 24, 21, 20]
```

```
average = _____/len(t)
print(f'avarage: average:.2f')
print('minimum:', _____)
print('maximum:', _____)
```

5.1.3

The *list()* function converts a given sequence of values into a list. The parameter must be **something that is iterable**, e. g.:

```
>>> list(range(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
>>> list()
[]
>>> a = ['milk','bread', 'butter','honey']
>>> b = list(a)
```

```
>>> b
['milk', 'bread', 'butter', 'honey']
```

In last example, the *b* list **is a copy** of the original *a* list.

Also when opening a file, result of the *open()* call can be transformed to a list of lines:

```
>>> list(open('input.txt', encoding='utf-8'))
['first line\n', 'second line\n', '\n', 'fourth line\n']
```

The following command would raise an exception, as the parameter (int number) is not iterable:

```
>>> list(2019)
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: 'int' object is not iterable
```

5.1.4

The *list()* function returns a new list with elements copied from the input iterable parameter.

- True
- False

5.1.5

In Python, **list are objects** of the list type and so provide some **methods** as well.

There are 2 methods that **do not change a list's content**:

```
list.count(x)
```

returns the number of times *x* appears in the list.

```
list.index(x)
```

returns the index of the *x* first occurrence (or raises a *ValueError* if there is no such item)

```
list.index(x, start, end)
```

returns the index of the *x* first occurrence, but the search is limited to a particular subsequence of the list (ranging from the *start* index to the *end-1* index).

In following examples, we put the corresponding outputs into the comments:

```
fruits = ['banana','apple','pier','cherry','banana','banana']
print(fruits.count('banana'))  # 3
print(fruits.index('banana'))  # 0
print(fruits.index('banana', 3, len(fruits)))  # 4
```

5.1.6

What will be printed out after executing the following script?

```
a = list('abcd') * 2

print(len(a))
print(a.count('a'))
print(a.index('b'))
```

- 8, 2, 1
- 4, 1, 1
- 8, 2, 2
- Nothing, because of a ValueError exception.

5.1.7

The other list methods are all **mutable** as they **modify lists' content** is some way.

For now, we take a look at methods for adding or removing items:

```
list.append(x)
```

adds an item *x* to the end of the list, its return value is *None*

```
list.insert(index, x)
```

inserts an item *x* at a given position, *index* parameter represents the position before which to insert, its return value is *None*

```
list.pop()
```

removes and returns the last item in the list

```
list.pop(index)
```

removes the item at the given position in the list and returns it

```
list.remove(x)
```

removes the first item from the list whose value is equal to x (or raises a *ValueError* if there is no such item)

```
list.clear()
```

removes all items from the list.

We will see some other list methods later.

In following examples, we put the corresponding outputs into the comments:

```
fruits = ['banana','apple','pier','cherry','banana','banana']
fruits.append('peach')
print(fruits)                # ['banana', 'apple', 'pier',
'cherry', 'banana', 'banana', 'peach']
print(fruits.pop())          # peach
print(fruits.pop(0))         # banana
fruits.insert(0, 'coconut')
fruits.remove('apple')
print(fruits)                # ['coconut', 'pier', 'cherry',
'banana', 'banana']
fruits.clear()
print(fruits)                # []
```

📖 5.1.8

Sometimes, we need to create **a new list by appending items to an empty list**:

```
numbers = []
for i in range(1000):
   if i % 3 == 0 and i % 5 == 0:
      numbers.append(i)
```

The list called *numbers* contains all integers lower than 1000 that are divisible by 3 and 5.

5.1.9

Consider the sequence of operations applied on a list.

Match the following lines with current content of the *a* variable that is being modified.

```
a = list(range(1, 6))
```

a == ____

```
a.remove(3)
```

a == ____

```
a.pop()
```

a == ____

```
a.insert(-1, 0)
```

a == ____

```
a.append(0)
```

a == ____

- [1, 2, 0, 4]
- [1, 2, 0, 4, 0]
- [1, 2, 4, 5]
- [1, 2, 3, 4, 5]
- [1, 2, 4]

5.1.10

Like for Python strings, we can get **slices** from lists, too.

To index a slice, we use the very same syntax:

```
list[start : stop : step]
```

Of course, each of the three parts written between **:** characters can be omitted. The step specification is optional (default value is 1).

The slicing operation **does not modify** the original list. **Each slice is a new list.**

Let us examine some examples:

```
>>> cities = ['Bratislava', 'Warsaw', 'Madrid', 'Praha']
>>> cities[1:3]
['Warsaw','Madrid']                    # from element at position
1 to the element at position 2
>>> cities[-3:]
['Warsaw','Madrid', 'Praha']           # from element with index
-3 to the end of the original list
>>> cities[:-1]
['Bratislava','Warsaw', 'Madrid']      # from beginning, to the
element before the last element (its index is -1)
>>> cities[1::2]
['Warsaw', 'Praha']                    # first element is at
index 1, the second at index 3
>>> cities[::-1]
['Praha', 'Madrid', 'Warsaw', 'Bratislava']    # all elements,
but in reversed order
```

5.1.11

Match the slices that would be created from the original list with their correct notations.

The original lists contains these names:

```
names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
```

____ == ['John']

____ == ['Francis', 'John', 'Luke', 'Mark', 'Matthew']

____ == ['Matthew', 'Luke', 'Francis']

____ == ['John', 'Francis']

____ == ['Matthew', 'Mark', 'Luke']

- names[::-1]
- names[:3:]
- names[3::]
- names[::2]
- names[3:4]

5.2 Processing Lists (programs)

5.2.1 Income I.

Input list contains 12 month earnings achieved by a student during the last year.

Find out, if he/she earned more during first or second half of the year.

```
Input : 1000,200,300,200,120,100,200,300,200,120,400,500
Output: first
```

```
Input : 0,0,0,200,100,100,100,300,200,100,400,0
Output: second
```

5.2.2 Income II.

Input list contains month earnings achieved by a student over a continuous sequence of months (starting from the beginning of a specific year).

Which months were the most successful ones? Print out a list of their names (not the indices).

```
Input : 1000,200,300,200,120,100,200,300,200,120,400,500
Output: ['january']
```

```
Input : 0,0,0,400,100,100,100,300,200,100,400,0,0,400
Output: ['april','november','february']
```

5.2.3 Cancelling Bus Lines

First list contains positive integer numbers identifying bus lines in a city.

Second list contains those numbers of bus lines that need to be cancelled (they have to be removed from the original list). Print out the first list after modifying it.

A bus line that we are trying to cancel may not exist. In such case, an error message has to be printed.

```
Input : 1,2,3,4,5,6,7,8
1,5,8
Output: [2,3,4,6,7]
```

```
Input : 1,2,3,4,5,6,7,8
1,2,9
```

```
Output: error
```

start.py

```
# do not modify this part of code:
input_list = input().split(',')
buses = []
if input_list != ['']:
  for x in input_list:
    buses += [int(x)]

input_list = input().split(',')
cancel = []
if input_list != ['']:
  for x in input_list:
    cancel += [int(x)]

# write your code here:
```

5.2.4 Removing All Occurences

First list contains positive integer numbers.

Second list contains those numbers from the original list that need to be removed (all of their occurrences).

Use some standard list functions in your solution. Print out the first list after modifying it.

```
Input : 8,1,1,1,1
1
Output: [8]
```

```
Input : 6,10,20,30,10,7,8
10,20,30
Output: [6,7,8]
```

start.py

```
# do not modify this part of code:
input_list = input().split(',')
a = []
if input_list != ['']:
  for x in input_list:
    aaa += [int(x)]

input_list = input().split(',')
```

```
b = []
if input_list != ['']:
  for x in input_list:
    b += [int(x)]

# write your code here:
```

5.2.5 Girls or Boys

Input list contains an even number of elements - names of girls and boys. On even positions, there are girls, on odd positions, there are boys.

For an input value 1, print out a new list containing all girls.

For an input value 2, print out a new list containing all boys.

Preserve the original order of elements.

```
Input : Anna,Peter,Martha,John,Andrea,Mike
1
Output: ['Anna','Martha','Andrea']
```

```
Input : Anna,Peter,Martha,John,Andrea,Mike
2
Output: ['Peter','John','Mike']
```

5.2.6 Cards

A "standard" deck of playing cards consists of 52 cards in each of the 4 suits of *spades*, *hearts*, *diamonds*, and *clubs*. Each suit contains 13 cards: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King.

There are 2 input lists prepared in this program. For a given suit and a given number of cards, print out the corresponding sequence of cards.

```
Input : hearts
3
Output: ['A hearts','2 hearts','3 hearts']
```

```
Input : diamonds
7
Output: ['A diamonds','2 diamonds','3 diamonds','4
diamonds','5 diamonds','6 diamonds','7 diamonds']
```

Lists as Parameters & Return Values

Chapter 6

6.1 Lists as Parameters & Return Values

🕮 6.1.1

By following assignment:

```
a = [10, 20, 30, 40]
```

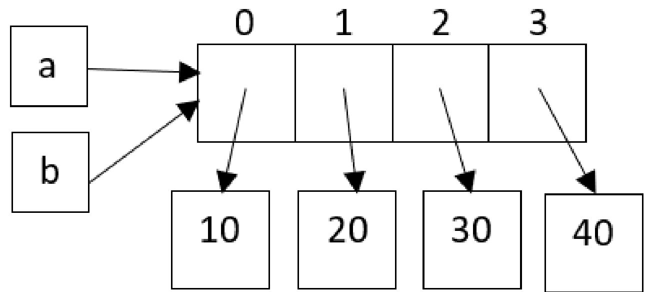
we set the *a* variable to reference a list with 4 elements.

One list can be referenced by multiple variables.

E. g. after the assignment:

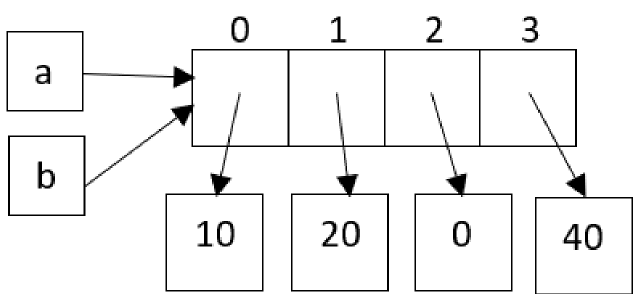
```
b = a
```

there are 2 variables referencing the same list:



This means, that when modifying the content of the *a* list, the *b* list is changed as well (it is the same list):

```
b[2] = 0
```



📝 6.1.2

Select all true statements:

- The list structure is mutable.
- Two different lists can be referenced with same variable at the same time.
- A list can be referenced with 10 different variables.
- After executing the assignments: a = [], b = [], both variables reference the same list in the memory.

📖 6.1.3

When getting a slice from a list, a new list is always created. The original list is not modified at all.

Slice notation can be used also at the left side of the assignment command. In this case, the right side must contain an expression returning some sequence of elements (not necessary a list). Such **assignment to a slice** is executed as follows:

- expression at the right side is evaluated and a new list is created
- the new list replaces a sublist defined by the slice

As the contents of list is modified, **assigning to a slice is a mutable operation**.

Watch carefully some examples:

```
>>> names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
>>> names[1:4] = ['Paul', 'Peter', 'Thomas']
>>> names
['Matthew','Paul', 'Peter', 'Thomas', 'Francis']
```

Three elements where replaced by another three.

```
>>> names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
>>> names[1:4] = ['Paul', 'Peter']
>>> names
['Matthew', 'Paul', 'Peter', 'Francis']
```

Three elements were replaced by two.

```
>>> names[-2:] = ['Andrew', 'James','Philip']
>>> names
['Matthew', 'Paul', 'Andrew', 'James', 'Philip']
```

The last two elements were replaced by three elements.

```
>>> names[:2] = []
>>> names
['Andrew', 'James', 'Philip']
```

The first two elements were deleted (this sublist was replaced by an empty list).

```
>>> names[:1] = 'Python'
>>> names
['P', 'y', 't', 'h', 'o', 'n', 'James', 'Philip']
```

The first element was replaced by six elements of a list constructed from a string.

6.1.4

Match each assignment to a slice with its result (current content of the list).

The original list contains 4 strings:

```
cities = ['Bratislava', 'Warsaw', 'Madrid', 'Praha']
```

Do not forget: While executing script shown below, the list *cities* is being modified:

```
cities[1:2] = []
```

cities == _____

```
cities[1:] = ['Budapest', 'Vienna']
```

cities == _____

```
cities[:] = range(1, 11)
```

cities == _____

```
cities[::2] = [0] * 5
```

cities == _____

```
cities[::] = []
```

cities == _____

- [0, 2, 0, 4, 0, 6, 0, 8, 0, 10]
- ['Bratislava', 'Madrid', 'Praha']
- ['Bratislava', 'Budapest', 'Vienna']
- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- []

🕮 6.1.5

Two lists can be compared using standard relational operators. Comparing is realized in a string-like way:

- the corresponding elements are compared one at a time until the first mismatch is found
- the result of comparing these elements is also the result of comparing the whole lists
- in case the lists are of different length, the shorter one is meant to be lower
- when comparing two elements, these have to comparable (e. g. operators < and > cannot be applied to compare an integer with a string etc., the *==* or *!=* operator would evaluate correctly)

Let us examine some examples:

```
>>> a = list(range(1,5))    #  [1, 2, 3, 4]
>>> b = list(range(1,5))    #  [1, 2, 3, 4]
>>> a == b
True
>>> a != b
False
>>> a[1] = 0
>>> a == b      # [1, 0, 3, 4] == [1, 2, 3, 4]
False
>>> a < b       # [1, 0, 3, 4] < [1, 2, 3, 4]
True
```

In the *b* list, the element with index 1 is greater than the corresponding one from the first list.

Some more comparisons:

```
>>> a <= b
True
>>> a > b
False
>>> a >= b
False
>>> [1,2] < b    # [1, 2] < [1, 2, 3, 4]
True
```

The first list with two elements that are equal to the corresponding elements of the *b* list, is shorter.

6.1.6

Select all expressions with result *True*:

- [1, 2, 3] <| [1, 2, 7]
- [1, 2, 3] >= [1, 2]
- [ ] == list()
- [1, 2, 3] <| [1, 2]
- [1, 2, 3] == ['mother', 'father', 'child']

6.1.7

When we need to process elements of lists, we often prepare a function **with a list parameter**.

The following function finds and returns the maximum value stored in the *a* list:

```
def maximum(a):
    m = a[0]
    for i in range(1, len(a)):
        if a[i] > m:
            m = a[i]
    return m

data = [1, 3, 8, 5, 6]
print(maximum(data))        # 8
print(data)                 # [1, 3, 8, 5, 6]
print(maximum('Python'))    # y
```

Notice, that when calling the function with a string (or some other iterable structure) as an actual parameter, everything is ok.

The function *maximum()* returned the greatest character (numerical codes of characters were compared in this case).

The second function serves for exchanging two elements in a list:

```
def exchange(a, i, j):
    if 0 <= i < len(a) and 0 <= j < len(a):
        a[i], a[j] = a[j], a[i]
    else:
        raise IndexError
```

```
data = [1, 3, 8, 5, 6]
exchange(data, 1, 3)
print(data)                  # [1, 5, 8, 3, 6]
```

The *a* list as **a formal parameter references the actual parameter** *data*.

So the *data* list is changed by the assignments realized within the function.

6.1.8

We want to write a function for deleting all elements from a given list.

Which of the two functions provides correct solution?

```
def clear1(list):
    list = []

def clear2(list):
    list.clear()

# consider results of the following calls
a = list('Python')
print(a)                    # ['P', 'y', 't', 'h', 'o', 'n']
clear1(a)
print(a)

a = list('Python')
print(a)                    # ['P', 'y', 't', 'h', 'o','n']
clear2(a)
print(a)
```

- clear2()
- clear1()

6.1.9

Besides using lists as parameters in functions, **a function can also return a list** of elements.

Consider a function for creating a new list with *n* elements initialized to a given value:

```
def create_list(n, value = 0):
    return [value] * n

print(create_list(10))        # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(create_list(5, -1))     # [-1, -1, -1, -1, -1]
print(create_list(8, None))   # [None, None, None, None, None,
None, None, None]
print(create_list(5, 'Hi'))   # ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

As in Python we do not need to specify an element's type, the *create_list()* function represents a general template for creating lists with elements set to some default value.

🕮 6.1.10

When writing functions that operate on lists, we must be careful. Sometimes, we want changes to by applied on the original list (provided in a parameter), but sometimes not!

Let us compare the following functions for adding a new element at the end of the list. Both of them return a list with one more element:

```
def add1(list, element):
    return list + [element]

def add2(list, element):
    list.append(element)
    return list

data = [1, 3, 8, 5, 6]

print(add1(data, 1000))         # [1, 3, 8, 5, 6, 1000]
print(data)                     # [1, 3, 8, 5, 6]
data = [1, 3, 8, 5, 6]
print(add2(data, 1000))         # [1, 3, 8, 5, 6, 1000]
print(data)                     # [1, 3, 8, 5, 6, 1000]
```

We have just seen, that the *add1()* function returns a new list and does not touch the actual parameter's content. The *add2()* function appends a new element onto the actual parameter.

So the *add1()* function is an **immutable** and the *add2()* function is a **mutable** operation!

6.1.11

We need to reverse the order of elements in a list.

Which of the following functions is a mutable operation?

```
def f1(a):
    return a[::-1]

def f2(a):
    a[::] = a[::-1]
    return a

def f3(a):
    b = []
    for x in a:
      b = [x] + b
    return b

# calling functions with same actual parameters
data = [1, 2, 3, 4, 5]
print(f1(data))
print(data)

data = [1, 2, 3, 4, 5]
print(f2(data))
print(data)

data = [1, 2, 3, 4, 5]
print(f3(data))
print(data)
```

- f1()
- f2()
- f3()

📖 6.1.12

There are two useful string methods concerning lists.

The *split()* method **returns a list** parsed from a given string .

The *join()* method takes **a list as a parameter** and returns a string.

At first about the splitting in more detail:

The *split()* method can be called without or with parameters (specification of a delimiter and a maximum number of splits is optional).

Compare the following 3 notations and the output for numbers typed by a user:

```
user_input = input('Enter 5 numbers: ').split()
print(user_input)
```

The input string returned by the *input()* function splits on any whitespaces if the delimiter parameter is left unspecified:

```
Enter 5 numbers: 10 20 30 40 50
['10', '20', '30', '40', '50']
```

```
user_input = input('Enter 5 numbers: ').split(',')
print(user_input)
```

The input string was splitted according the specified delimiter (the comma character):

```
Enter 5 numbers: 10,20,30,40,50
['10', '20', '30', '40', '50']
```

```
user_input = input('Enter 5 numbers: ').split(',', 2)
print(user_input)
```

The *split()* method returned a list with 3 elements as the numbers of splits is set to 2.

```
Enter 5 numbers: 10,20,30,40,50
['10', '20', '30,40,50']
```

🕮 6.1.13

The *join()* method returns a string, which is the concatenation of strings in the list provided by a parameter. The separator between list elements is the string providing this method.

Study the following examples:

```
words = ['sun', 'is', 'shining']
message = ' '.join(words)
print(message)                    # sun is shining

import random
numbers = [str(random.randint(1, 9)) for i in range(5)]
print(numbers)                    # e. g. ['9', '1', '9',
'3','2']
excercise = '+'.join(numbers)
print(excercise,' = ')            # 9 + 1 + 9 + 3 + 2 =
```

6.1.14

The user will type a sentence in a console and the program will respond by printing an Output:

```
output = ' '.join(input('enter your sentence: ').split()[::-
1])
print(output)
```

What is the output for the following input sentence?

The Earth is my home.

- home. my is Earth The
- The Earth is my home.
- The-1Earth-1is-1my-1home.
- The first line of code is incorrect, an exception would be raised.

6.2 Lists as Parameters & Return Values I. (programs)

6.2.1 Counting Occurances

Write function *my_count()* with two arguments - a list of integer values and a value to find. Calculate number of the given value's occurances.

```
Input : 1,1,3,2,1,0,3
3
Output: 2
```

```
Input : 1,1,3,2,1,0,2
5
Output: 0
```

6.2.2 Looking for a Position

Write function *my_index()* with two arguments - a list of integer values and a value to find. Return the leftmost position of the given value or raise the *ValueError* exception if there is no such value.

```
Input : 1,1,3,2,1,0,3
3
Output: 2
```

```
Input : 1,1,3,2,1,0,2
5
Output: ValueError
```

6.2.3 Counting Inversions

Let us assume the following input sequence of integer values:

$a_0, a_1, ..., a_i, ..., a_n$

For the a_i element, the number of inversions equals to number of such a_k elements, that $k < i$ and $a_k > a_i$.

Write function *number_of_inversions()* with one argument - a list of integer values.

Return the total number of inversions found while processing all of the values in a given list.

```
Input : 1,5,6,3,4,7,0
Output: 10
```

```
Input : 1,2,3,4,5
Output: 0
```

6.2.4 Reversing a Sequence

Write 2 versions of a function for reversing the order of values stored in a list.

The *my_reverse1()* function must be implemented as *immutable* operation.

The *my_reverse2()* function must be implemented as *mutable* operation.

You can assume the following testing code:

```
print(str(my_reverse1(a)))
print(a)
my_reverse2(a)
print(a)
```

The examples of inputs and outputs:

```
Input : 1,2,3,4,5
Output: [5,4,3,2,1]
[1,2,3,4,5]
[5,4,3,2,1]
```

```
Input :
Output:[]
[]
[]
```

6.2.5 Inserting into an Ordered Sequence

Write function *my_insert()* with 2 arguments - an ordered list of integer values and a value to be inserted into the list. The updated list is returned by the function.

The input list is ordered in ascending order. After inserting a new value, the list remains sorted.

```
Input : 1,2,4,5
3
Output: [1,2,3,4,5]
```

```
Input :
7
Output: [7]
```

```
Input : 2,4,5,7
0
Output: [0,2,4,5,7]
```

6.2.6 Election Winner

The elections are over. The candidates were numbered with integers from *1* to *n*. The list *a* contains choices of all people that have participated actively by choosing one of the *n* candidates.

Write function *winner_of_elections()* with 2 arguments - the number of candidates and list of choices made by people. The function returns number of the winning candidate.

In case the winner cannot be decided exactly, raise a *ValueError* exception (the testing program will print out the appropriate string for you)..

```
Input : 10
1,1,2,1,1,1,1,1,3,5,5,7,8,9
Output: 1
```

```
Input : 10
8,8,8,8,8,8,8,8,8
Output: 8
```

```
Input : 3
1,1,1,3,2,2,2,3
Output: The elections must be repeated.
```

6.2.7 Merging Two Ordered Sequences

Write function *merge()* with 2 arguments - 2 lists of integer values. The input lists are ordered in ascending order.

The function returns a new ordered list containing all elements from both input lists. The resulting list is also ordered in ascending order.

```
Input : 1,2,4,5,10
3,6,7
Output: [1,2,3,4,5,6,7,10]
```

```
Input :
1,8,12
Output: [1,8,12]
```

6.3 Lists as Parameters & Return Values II. (programs)

6.3.1 From File to List

Write function *from_file_to_list()* with 1 argument - the name of a text file containing integer values (one value per line).

The function returns a list contaning all numbers from the input file.

```
Input : data1.txt
Output: [1,2,3,4,5,6,7,8]
```

```
Input : data2.txt
Output: [10,20,30]
```

```
data1.txt:
1
2
3
4
5
6
7
8
```

```
data2.txt:
10
20
30
```

6.3.2 Union and Intersection

Write 2 functions for creating the union and intersection of two input lists.

The function *union()* takes two input lists as arguments and returns a new one containing all of the values found in either of them.

The function *intersection()* takes two input lists as arguments and returns a new one containing all of the values found in both of them.

Be careful about order of elements in the resulting list. We are intersted only in unique values, so any duplicity must be filtred.

```
Input : 1,1,2,4,5,10
2,3,6,7,10
Output: [1,2,4,5,10,3,6,7,10]
[2,10]
```

```
Input :
1,8,12
Output: [1,8,12]
[]
```

6.3.3 Travelling the World I.

We are travelling around the world. Cities are numbered with integers from 1 to n. A sequence of numbers represents a route that we have made, e. g. when we start in city 1 and continue to cities 2, 5 and 8, the input list would be: [1,2,5,8].

Sometimes, we can visit the same city more than once. In such situations, we say, that the route contains a circle.

Write function *check_for_circle()* with one argument - a list of integer values representing a sequence of cities visited while travelling.

The function returns *True*, if a circle was detected, *False* otherwise.

```
Input : 1,2,3,4,5,6,3,2,1,2
Output: True
```

```
Input : 1,2,3,4,5,8
Output: False
```

6.3.4 Travelling the World II.

We are travelling around the world. Cities are numbered with integers from 1 to n. A sequence of numbers represents a route that we have made, e. g. when we start in city 1 and continue to cities 2, 5 and 8, the input list would be: [1,2,5,8].

Sometimes, we can visit the same city more than once. In such situations, we say, that the route contains a circle.

Write function *check_for_circle()* with one argument - a list of integer values representing a sequence of cities visited while travelling.

The function returns a new list representing the first detected circle. In case, the input sequence does not contain any circles, an empty circle is returned.

```
Input : 1,2,3,4,5,6,3,2,1,2
Output: [3,4,5,6,3]
```

```
Input : 1,2,3,4,5,8
Output: []
```

6.3.5 Merging with Slices

For two lists of integers, create and print a new list of integers constructed as follows:

On *even positions*, the resulting list contains elements from the first input list that are also on even positions. Their order is preserved.

On *uneven positions*, the resulting list contains elements from the second input list. Be careful, their other is reversed.

Assume, that the second list always contains exactly the necessary number of elements.

Notice, that the new list is of the same length as the first input list.

```
Input : 1,0,1,0,1,0
2,3,4
Output:[1,4,1,3,1,2]
```

```
Input : 5,5,23,5,80,5,78,5,3
1,0,0,0
Output:[5,0,23,0,80,0,78,1,3]
```

```
Input : 3

Output:[3]
```

```
Input :

Output: []
```

6.3.6 Comparing Lists of Words

Compare two lists of words and print out, whether they are the same or not.

In case the inpust lists are not equal, print out the length of matching sublists.

```
Input : alex,brandon,cecila,daniel
alex,brandon,tom
Output: first 2 words are the same
```

```
Input : windows,linux,mac os,android
windows,linux,mac os,android
Output: same sequences of words
```

```
Input : a
a,b,c
Output: first word is the same
```

```
Input : hogo
fogo
Output: first 0 words are the same
```

⌨ 6.3.7 Transforming Program's Notation

We program a robot to move in either of the 4 directions using s set of commands:

n (north), *s* (south), *e* (east) and *w* (west).

The input string (watch the examples below) represents a program. The -#- character serves as a delimiter.

To make code more readable, print out a new notation representing the same program:

1. In output string, the -#- delimiters are replaced with spaces.
2. Instead of writing the same command repeatedly, the number of repeats is written before the command itself.

```
Input : nnnnn#wwww#sss#ww#nn
Output: 5*n 4*w 3*s 2*w 2*n
```

```
Input : s
Output: 1*s
```

⌨ 6.3.8 Shopping

In text files, there ara data about things bought while shopping in a mall.

Each line contains 3 or 4 pieces of data separated by semicolons:

- item name
- number of items bought
- price of 1 item
- ! for 10 % discount

For an input text file given by its name, calculate total sum to pay.

```
Input : data1.txt
Output: 67.20
```

```
Input : data2.txt
Output: 2.00
```

```
data1.txt:
a;10;5;!
b;5;1
c;1;12.5
d;2;2.35
```

```
data2.txt:
a;1;1
b;1;1
```

6.3.9 Offline Status

In text files, there ara data about users of a social network.

Each line contains multiple pieces of data separated by semicolons:

- person-s name
- dates or periods when the user was offline the whole day

Print out the name of a person with minimum number of offline days.

Be careful about the input lines: When a user's offline period exceeds over the following month, a new record is always made, e.g. 15.1.-31.1.;1.2.-5.2. means that some user was offline continuosly from 15.1. to 5.2.

```
Input : data1.txt
Output: Paul
```

```
Input : data2.txt
Output: Linda
```

```
Input : data3.txt
Output: empty file
```

```
data1.txt:
John;1.5.-17.5.;7.6.
Paul;8.3.;12.3.;15.6.-18.6.
Michael;2.2.-20.2.
Peter;1.5.;8.6.;12.12.;14.12.-31.12
```

```
data2.txt:
Maria;1.1.;7.10.-31.10;1.11.-2.11.
Linda;18.5.
Julia;3.5.-18.5.;12.12.-20.12.
```

```
data3.txt:
```

6.3.10 Maths Exercises

In text files, simple mathematical exercises for young pupils are prepared. Only positive integers as operands and 4 basic arithmetic operations are allowed.

The non-empty input list contains results of all the exercises calculated by a pupil.

For an input text file given by its name and a list with a pupil's results, count all the correct answers.

```
Input : data1.txt
15,0,140,5
Output:  4
```

```
Input : data1.txt
15,1,140,5
Output:  3
```

```
data1.txt:
10+5
1-1
7*20
15/3
```

Functions as Values

Chapter 7

7.1 Functions as Values

🕮 7.1.1

Python belongs to those programming languages that support also **functional programming style**.

Functions can be

- passed as arguments to other functions,
- returned from other functions as well as
- assigned to variables or stored in data structures.

Let us define 3 functions:

```
def poly1(x):
    return x ** 2 - 2 * x + 1

def poly2(x):
    return x ** 2 + 2 * x + 1

def poly3(x):
    return x ** 2 - 1
```

The name of a function is a value, that can be assigned to variables, e. g.:

```
fun = poly1
```

From now on, the function *poly1()* can be called in either of the 2 ways:

```
fun = poly1      # the fun variable references the poly1
function
a = poly1(10)    # calling the poly1 function directly
b = fun(10)      # calling the poly1 function, using a variable
print(a, b)      # 81 81
```

The *fun* variable is of a functional type:

```
print(type(fun))     # <class 'function'>
```

##

One function can be assigned to more variables.

- True
- False

7.1.3

Very often, more functions (references to them) are stored in a data structure, e. g. a list.

When we have references to functions in a list, functions can be accessed using indices:

```
def poly1(x):
    return x ** 2 - 2 * x + 1

def poly2(x):
    return x ** 2 + 2 * x + 1

def poly3(x):
    return x ** 2 - 1

funs = [poly1, poly2, poly3]
x = funs[2](10)    # calling the poly3 function
print(x)           # 99
```

In the following example, we choose the index randomly. The selected function is called with 0 as an actual argument.

```
x = funs[random.randrange(len(funs))](0)
print(x)    # 1 or -1 dependig on the randrange() result
```

7.1.4

The following program was executed and the output was „a smiling face“.

Complete the last line of code correctly.

```
def sad(): return ':-('
def happy(): return ':-)'
def surprised(): return ':-O'

moods = [sad, happy, surprised]
print(_____)
```

🕮 7.1.5

Let us consider some simple functions:

```
def f1(x):
    return x ** 2

def f2(a, b, c):
    return a ** 2 + b ** 2 == c ** 2

def f3(x):
    return x % 100
```

We call the functions using their names:

```
print(f1(7))          # 49 (power of 7)
print(f2(3, 4, 5))    # True (3, 4, 5 are sides of
a rectangular triangle)
print(f3(2015))       # 15 (last two digits)
```

Each of the functions evaluated an expression and returned the calculated value.

🕮 7.1.6

Quite often, such simple functions (having just one line) are not needed to be named. We can implement them as **anonymous** using the **lambda** construction (writing an **lambda expression**):

```
fun = lambda x: x ** 2
print(fun(7))                                       # 49

fun = lambda a, b, c: a ** 2 + b ** 2 == c ** 2
print(fun(3, 4, 5))                                 # True

fun = lambda n: n % 100
print(fun(2015))                                    # 15
```

Notice the general syntax we use when writing lambda expressions.

Instead of implementing the body of a function:

```
def function_name(parameters):
    return expression
```

we define an anonymous function:

```
lambda parameters : expression
```

In our testing script, the *fun* variable was used as an auxiliary variable to reference the individual lambda expressions.

Usually, we do not store references to lambda expressions as they are used directly at the place, where we write them.

Later, we will see lambda expressions applied when passing anonymous functions as arguments to other functions.

7.1.7

Complete the lambda expression equivalent to the following function:

```
def fun(x, y):
   return x * y
```

```
f = lambda _____ : x * y

print(f('Python', 2))    # PythonPython
print(f(7, 5))           # 35
print(f([0], 3))         # [0, 0, 0]
```

7.1.8

We have already seen, that **functions in Python are treated as values** (they can be stored in variables, returned from functions as well as passed as arguments to other functions). Such first-class functions are a necessity for the functional programming style, in which the use of higher-order functions is a standard practice.

A simple example of a **higher-ordered function** is the standard *map()* function.

The *map()* function takes, as its arguments, a **function** and a list (or some other iterable structure), and returns an iterable object formed by applying the function to each element of the list.

The object returned from the *map()* function can be transformed into a list easily.

Let us explore the following situation:

```
def my_function(str):
    n = 0
    for c in str:
        if c in 'aeiouy':
            n += 1
    return n

a = 'The Big Bang Theory'.split()
print(a)                              # ['The', 'Big', 'Bang', 
'Theory']
print(list(map(len, a)))              # [3, 3, 4, 6]
print(list(map(my_function, a)))      # [1, 1, 1, 3]
print(list(map(lambda x: x[0], a)))   # ['T', 'B', 'B', 'T']
```

First, for each element of the *a* list, its length is calculated. The first argument of the *map()* function is the standard *len()* function.

Then, for each element of the *a* list, the *my_function()* counts number of vowels. The *my_function()* is passed to the *map()* function as the first argument.

Finally, for each element of the *a* list, its first character is returned. The lambda expression represents an anonymous function passed to the *map()* function.

For mapping purposes, we could also write our own functions, e. g.:

```
def my_map(fun, a):
    return [fun(element) for element in a]
```

Compare the following lines of code:

```
print(my_map(lambda x: x ** 2, [1, 2, 3]))      # [1, 4, 9]
print(list(map(lambda x: x ** 2, [1, 2, 3])))   # [1, 4, 9]
```

In both cases, we passed an anonymous function using lambda expression into the mapping function as its first argument.

The *my_map()* function could return the list type object directly, so we did not need to transform it afterwards.

7.1.9

Here is an example of using the standard *map()* function in action:

```
user_input = input('Your numbers:')
a = list(map(int, user_input.split()))
print(a)
```

Select all true statements:

- When user inputs 3 integer numbers separated with spaces, the a will be a list containing 3 strings.
- When user inputs 3 integer numbers separated with spaces, the a will be a list containing 3 ints.
- The result of the input() call is a string.
- The first argument of the map() function is an anonymous function.

🕮 7.1.10

The *map()* function is often used to transform a list of elements to another list of elements.

But, the resulting object can be iterated also directly:

```
a = ['The', 'Big', 'Bang', 'Theory']
for e in map(lambda x: x[0], a):
    print(e, end=' ')                     # T B B T
```

Thanks to the lambda expression, from each of the string elements, only the first character was selected.

🕮 7.1.11

The second argument of the *map()* function could be of any other iterable type. Watch the examples:

```
b = range(2015, 2020)
print(b)                                   # range(2015,2020)
represents the sequence: 2015,2016,2017,2018,2019
print(list(map(lambda x: x % 10, b)))      # [5, 6, 7, 8, 9]
```

The elements in *b* were mapped to their last digit.

```
print(list(map(int, str(2 ** 10))))       # [1, 0, 2, 4]
```

The number 2 ** 10 == 1024 was converted to a string '1024' at first in order to map the individual characters to decimal digits (ints).

7.1.12

What does the *mystery()* function do?

```
def mystery(x):
    return sum(map(int, str(x)))

print(mystery(2019))
```

- It sums all digits of a given number, so the result is 12.
- Converts a given number to a list of its digits, so the result is [2,0,1,9].
- It converts a given number into a string by concatenating the '2', '0', '1', '9'.

🕮 7.1.13

Sometimes, we need to remove all elements from a given list that satisfy a specific condition.

We could prepare the filtering function like this:

```
def my_filter(fun, a):
    return [element for element in a if fun(element)]

numbers = [48, 5, 7, 1, 22, 10, 13, 80]
print(my_filter(lambda x: x % 2 == 0, numbers))   # [48, 22, 10, 80]
```

The *my_filter()* function returned a new list containg the even numbers. Why?

An anonymous function (the lambda expression) was passed into the *my_filter()* function and the formal parameter *fun* was set to reference it.

For uneven numbers, the *fun* function returns *False* while constructing the new list, so the uneven numbers were filtered correctly.

For filtering purposes, we can use the standard *filter()* function.

Like with *map()* function, the resulting object is iterable and can be converted to a list easily:

```
numbers = [48, 5, 7, 1, 22, 10, 13, 80]
print(list(filter(lambda x: x % 2 == 0, numbers)))      # [48, 22, 10, 80]
```

First argument of the *filter()* function is a function representing the filtering condition. It has one parameter and returns *True* or *False*. In our case, an anonymous function (the lambda expression) was used.

7.1.14

Check all the elements of the original list that are in the resulting list.

```
passwords = ['abcd123', 'oxo', 'XyZ1111', 'fun3']
print(list(filter(lambda x: len(x) > 5, passwords)))
```

- 'abcd123'
- 'oxo'
- 'XyZ1111'
- 'fun3'

7.1.15

Very often, elements stored in a list have to be sorted in some specific order.

In Python, the *sorted()* function and the *sort()* method are available.

The sorting with *sorted()* and *sort()* are another examples of **passing anonymous functions** (the lambda expressions) **as arguments** to another functions.

The *sorted()* function takes a list as its argument and **returns a new list** with elements sorted in the ascending order, e. g.:

```
a = [1, 7, -2, 10, 13, 8, 41, 3]
print(sorted(a))                    # [-2, 1, 3, 7, 8, 10,
13, 41]
print(a)                            # [1, 7, -2, 10, 13, 8,
41, 3]
```

The original list remained unsorted.

The *sorted()* function has two optional parameters – **reverse** and **key**.

We can require the reversed order like this:

```
print(sorted(a, reverse = True))
```

```
print(a)                                    # [41, 13, 10, 8, 7, 3,
1, -2]
```

We can specify the sorting criterion like this:

```
print(sorted(a, key = lambda x: x**2))
print(a)                                    # [1, -2, 3, 7, 8, 10,
13, 41]
```

The *key* parameter is a function of one argument that transforms each element to its key while sorting. **Instead of elements, their keys are being compared**. The default velue is *None* (compare the elements directly).

In our case, the numbers were ordered according to their powers.

7.1.16

The following list was sorted using the *sorted()* function. What is the correct output?

```
a = [3, 4, 1, 2]
b = sorted(a, reverse = True)
print(a, b)
```

- [3, 4, 1, 2] [4, 3, 2, 1]
- [4, 3, 2, 1] [4, 3, 2, 1]
- [3, 4, 1, 2] [1, 2, 3, 4]

7.1.17

In fact, the *sorted()* function can be applied on any iterable structure, e. g. on strings as well:

```
msg = 'Python'
print(sorted(msg))                                  # ['P', 'h',
'n', 'o', 't', 'y']
print(sorted(msg, reverse = True))                  # ['y', 't',
'o', 'n', 'h', 'P']
print(''.join(sorted(msg, key = str.lower)))        # hnoPty
```

In the very last line of code, the characters were converted to lower case before comparing. We used the string *lower()* function as the optional *key* parameter.

7.1.18

The *sorted()* function is applicable also for tuples.

The output of calling:

```
print(sorted((7,2,4,1)))
```

would be

```
[1, 2, 4, 7]
```

- True
- False

7.1.19

Calling the *sort()* method on a list is **a mutable operation**.

This method sorts the original list and returns *None*, e. g.:

```
years = [1980, 1969, 1971, 1978, 2015, 1945, 2012, 1984]
print(years.sort(key = lambda x: x % 100, reverse = True))    #
None
print(years)                                                  #
1984, 1980, 1978, 1971, 1969, 1945, 2015, 2012]
```

The years are sorted according to their final two digits. The order is reversed.

7.1.20

What is the output of the following program?

```
names = ['Mathew', 'John', 'Mike', 'Peter', 'Ela', 'George']
names.sort(key = len)
print(names)
```

- ['Ela', 'John', 'Mike', 'Peter', 'Mathew', 'George']
- ['Mathew', 'John', 'Mike', 'Peter', 'Ela', 'George']
- [3, 4, 4, 5, 6, 6]

- None of the options, because the first letters of names do not correspond with the alphabetic order.

7.2 Functions as Values (programs)

7.2.1 Years of Birth

Input list of positive integers represents the birth years of people attending the same event.

Sort the elements of an input list according to their last two digits.

Print out the list after sorting.

```
Input : 1969,2003,1978,1980,2015
Output: [2003,2015,1969,1978,1980]
```

```
Input : 1999,2098
Output: [2098,1999]
```

7.2.2 Various Criteria for Sorting

Input list of strings may be sorted applying different criteria. These criteria are saved in a list as lambda expressions (anonymous functions):

```
criteria = [lambda x: x, lambda x: x[::-1], lambda x: len(x)]
```

Finish the program for sorting strings according to the following requirements:

- read the input list of strings,
- read an integer number representing a criterion choice,
- print out the sorted list.

```
Input : aaay,bbz,cx
0
Output: ['aaay','bbz','cx']
```

```
Input : aaay,bbz,cx
1
Output: ['cx','aaay','bbz']
```

```
Input : aaay,bbz,cx
2
```

```
Output: ['cx','bbz','aaay']
```

7.2.3 Questionnaire Data

Input text files contain data gathered from questionnaires that were filled by many respondents.

Respondents were choosing answers by checking an integer number on a scale from 0 to 6. In each line, there are a respondent's choices to n questions. The questions are numbered from 0 to n-1.

Read an input file's name and a question's number. Process the input file to find out how many respondents chose the neutral answer 3 for a given question.

```
Input : data1.txt
1
Output: 4
```

```
Input : data1.txt
6
Output: 0
```

```
data1.txt:
0;3;5;4;3;5;5
4;4;0;2;3;3;5
0;3;0;2;3;1;5
2;3;0;2;2;1;5
0;3;0;0;0;1;0
```

7.2.4 Multiplication of Digits

Write function *product_of_digits()* with one argument - an integer number.

The function multiplies all the input number's digits and returns the resulting product.

```
Input : 2019
Output: 0
```

```
Input : 1234
Output: 24
```

7.2.5 Filtering User Names

Read a list of users' names (strings). Modify this list to contain only strings longer than 5 characters.

The list has to be ordered in descending order after filtering shorter names.

Print out the list's status after modification.

```
Input : John,Max,Alexander,Francis,Robert
Output: ['Robert', 'Francis', 'Alexander']
```

```
Input : abrakadabra,bububu
Output: ['bububu','abrakadabra']
```

7.2.6 Primes

Correct and finish the following program:

```
def is_prime(x):
    pass

numbers = input()
print(filter(is_prime(x), numbers))
```

For a non-empty input list of positive integers, print out a new list containing only prime numbers.

Do not change the order of elements.

```
Input : 48 5 7 1 22 10 77 13 80
Output: [5, 7, 13]
```

```
Input : 2 27 120
Output: []
```

Tuples

Chapter 8

8.1 Tuples

📖 8.1.1

Tuple is another important structured type in Python.

Tuples are very similar to lists as they are **ordered collections of elements** too. But unlike lists, tuples are **immutable**. Once a tuple is created, it cannot be changed.

Tuples are written with round brackets. Elements are separated with commas, napr.:

```
>>> point = (100, -75)
>>> point
(100, -75)
>>> type(point)
<class 'tuple'>
```

The *point* variable references a tuple of 2 values (its coordinates).

Using the *type()* function, we can determine type of variable easily.

📖 8.1.2

Here are some other examples of working with tuples:

```
>>> team = ('John', 'Peter', 'Paul', 'Michael', 'George')
>>> team
('John', 'Peter', 'Paul', 'Michael', 'George')
>>> print(len(team))
5
>>> print(team[1])
Peter
```

In above example, we created a tuple of 5 strings (players). We printed out the tuple’s length as well as the item *team[1]*. We applied the standard *len()* function and indexing.

```
>>> order = ('T-Shirt', 'red', 'XL', 12.0, 1000, True)
>>> order[2] = 'M'
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The tuple *order* consists of 6 elements that are of different types. We tried to change the element at index 2, but this operation is not supported (as tuples are immutable).

8.1.3

By calling the *tuple()* function, we can make tuples from other sequences, e. g. strings, lists or ranges:

```
>>> tuple('Python')
('P', 'y', 't', 'h', 'o', 'n')
>>> tuple([2, 3, 5, 7])
(2, 3, 5, 7)
>>> tuple(range(1, 11))
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

8.1.4

Check all the correct statements:

- To access elements of tuples, we use indices.
- The length of a tuple cannot be changed.
- In tuples, all elements must be of a same type.
- The length of a tuple cannot be printed out.

8.1.5

Let us take a look at some special cases of tuples:

It is possible to define an **empty tuple**. We can call *tuple()* or write the **()** directly. The length of an empty tuple is 0:

```
>>> tuple()
()
>> ()
()
>> len(())
0
```

Sometimes, we may get an empty tuple as a result of calling a function.

📖 8.1.6

Tuples consisting of **the only one element** have to be written with a comma:

```
>>> melody = ('C',)
>>> melody
('C',)
>>> len(melody)
1
>>> type(melody)
<class 'tuple'>
```

Leaving the comma out, the value would be of a different type:

```
>>> melody = ('C')
>>> melody
'C'
>>> type(melody)
<class 'str'>
```

Be careful when working with tuples:

```
>>> (3.14)
3.14
>>> (3,14)
(3, 14)
```

First, we used decimal point and a number was created. Then, we did a typo. Comma after the first digit was interpreted as a separator of two elements in a tuple.

📝 8.1.7

Check the correct definitions of tuples:

- ()
- (7)
- (5,7)
- (5.3)

🕮 8.1.8

The **immutable operations** we use for lists are applicable on tuples as well.

Two tuples can be concatenated using the + operator:

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
```

The multiple concatenation is possible as well:

```
>>> (1, 0) * 8
(1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0)
```

With the *in* operator, we test the presence of an element:

```
>>> melody =  ('c', 'd', 'e', 'f', 'g')
>>> 'h' in melody
False
```

Slicing is also supported, e. g.:

```
>>> melody[:2]
('c', 'd')
>>> melody[::-1]
('g', 'f', 'e', 'd', 'c')
>>> melody[3:]
('f', 'g')
```

Methods for counting or finding the elements are available:

```
>>> melody = tuple('cdefg')
>>> melody
('c', 'd', 'e', 'f', 'g')
>>> melody.count('e')
1
>>> melody.index('g')
4
>>> melody.index('h')
Traceback (most recent call last): File "<input>", line 1, in
<module>
ValueError: tuple.index(x): x not in tuple
```

The standard *len()*, *sum()*, *min()*, *max()* functions are sometimes useful too.

Tuples are comparable with relational operators (==, !=, <, <=, >, >=) like lists. But the corresponding elements of tuples must be comparable.

```
>>> ('Peter', 'Bum', 1970) < ('Peter', 'Bim', 2000)
False
```

8.1.9

Tuples are **iterable**:

```
primes = (2, 3, 5, 7, 11, 13)
for i in primes:
    print(i, end=' ')      # 2 3 5 7 11 13
```

Sometimes, we need to enumerate all elements of a tuple within the *for* statement. In this case, the round brackets are not necessary:

```
for i in 2, 3, 5, 7, 11, 13:
    print(i, end=' ')
```

Tuples are often used in multiple assignments, e. g.:

```
x, y, z = 10, 20, 30
print(x, y, z)             # 10, 20, 30
```

The values on the right side separated with commas make up a tuple of length 3.

8.1.10

Remember that **tuples are immutable**, so this is not allowed:

- using *del* operator (elements cannot be removed),
- adding new elements,
- assigning new values to existing elements.

8.1.11

Match all the *print()* statements with their correct outputs.

```
u = (10, 20, 30)
v = (-10, -20, -30)
```

```
print(u + v)
```

____

```
print(u * 3)
```

____

```
print(u[:1] + (0,) + v[-1:])
```

____

- (30,40,60)
- (10, 0, -30)
- (10, 20, 30, -10, -20, -30)
- (0, 0, 0)
- (10, 20, 30, 10, 20, 30, 10, 20, 30)

📖 8.1.12

In Python, it is possible to return more than one value from a function. **When returning multiple values, they are all packed into a tuple**.

The return values have to be separated with commas, the round bracket are not required.

The following example shows a function that returns 2 random numbers:

```
def my_fun(a, b):
    x = random.randrange(a, b)
    y = random.randrange(a, b)
    return x, y
```

The returned tuple can be used as is, or assigned into the corresponding numbers of variables:

```
print(my_fun(1, 10))        # e. g. (7, 4)
m, n = my_fun(1, 10)        # e. g. m, n = (2, 5)
print(m, n)                 # 2 5
```

8.1.13

When we do not need all return values, we can use indices.

The function *div_mod()* returns 2 values packed in a tuple:

```
def div_mod(a, b):
    return a // b, a % b
```

First, we will print out the division, after that, the remainder:

```
result = div_mod(7, 2)
print(result)                        # (3, 1)
print('division:', result[0])        # division: 3
print('remainder:', result[1])       # remainder: 1
```

8.1.14

The following example shows that tuples returned from a function can be processed as needed:

This function determines all divisors of a number n.

```
def divisors(n):
    d = ()
    for i in range(1, n + 1):
        if n % i == 0:
            d = d + (i,)
    return d
```

The returned tuple can be printed out or used in further computations:

```
print(divisors(12))       # (1, 2, 3, 4, 6, 12)
for i in divisors(12):
    print(i, end=' ')     # 1 2 3 4 6 12

x = 13
if len(divisors(x)) == 2:
    print(f'{x} is a prime number')     # 13 is a prime number
```

8.1.15

The following function takes an integer number and returns a tuple of its divisors:

```
def divisors(n):
    d = ()
    for i in range(1, n + 1):
        if n % i == 0:
            d = d + (i,)
    return d

result = divisors(12)
```

Match the *print()* statements with their corresponding outputs:

```
print(result)
```

____

```
print(len(result))
```

____

```
print(2 in result)
```

____

```
print(sum(result[1:-1]))
```

____

- 15
- True
- 6
- (1, 2, 3, 4, 6, 12)

🕮 8.1.16

A **variadic function** accepts a variable number of arguments. In Python, we can use tuples to write such functions.

At first, consider the following function for multiplication. It has 4 arguments. There are default values set for the last 3 of them.

```
def multiply(a, b=1, c=1, d=1):
    return a * b * c * d
```

Now, the *multiply()* function can be called with 1, 2, 3 or 4 arguments:

```
print(multiply(2))                   # 2
print(multiply(2, 3))                # 6
print(multiply(2, 3, 5))             # 30
print(multiply(2, 3, 5, 2))          # 60
# print(multiply(2, 3, 5, 2, 7))     # 320? no, this is not
allowed
```

The *multiply()* function seems to be quite flexible, but the number of arguments is limited to 4.

We would like to pass really a variable number of arguments into the function. To make it possible, we have to use a packed argument when implementing the function. The **packing operator** * must be written just before the packed argument, it is the *numbers* in this case:

```
def multiply(*numbers):
    m = 1
    for n in numbers:
        m *= n
    return m

# calling  a variadic function (passing 1, 3 and 8 arguments)
print(multiply(1))                           # 1
print(multiply(4, 8, 2))                     # 64
print(multiply(1, 2, 3, 4, 5, 6, 7, 8))   # 40320
```

All the actual arguments were **packed in a tuple**. In function, we accessed the elements of this tuple.

8.1.17

Complete the head of the following function correctly:

```
def welcome_everybody(shout, _____):
    for p in persons:
        if shout:
            print('HALLO ', p, '!')
        else:
            print('Hallo ', p, '!')

# calling the function with various numbers of arguments
welcome_everybody(True, 'Susan')
welcome_everybody(True, 'Susan', 'Kate' )
```

```
welcome_everybody(False, 'Peter', 'John', 'Joseph', 'George',
'Thomas')
```

8.1.18

The *get_random_items()* function illustrates a situation of having 1 required positional argument followed by an unknown number of other arguments:

```
def get_random_items(n, *choices):
    items = ()
    for i in range(n):
        index = random.randrange(len(choices))
        items = items + (choices[index],)
    return items
```

The above function **returns a tuple** that is made up of *n* elements randomly chosen from the *choices* argument.

Let us call this function with various arguments. We would like to get:

- 2 numbers randomly chosen from 0, 1, -1,
- 5 numbers randomly chosen from 0, 1, -1,
- 1prime number randomly chosen from the 6 given options,
- 8 characters randomly chosen from 'R', 'G', 'B.

```
print(get_random_items(2, 0, 1, -1))              # e. g. (1, 0)
print(get_random_items(5, 0, 1, -1))              # e. g. (1, 1,
0, 1, -1)
print(get_random_items(1, 2, 3, 5, 7, 11, 13))   # e. g. (7,)
print(get_random_items(8, 'R', 'G', 'B'))         # ('B', 'G',
'B', 'G', 'R', 'R', 'B', 'B')
```

8.1.19

Let us examine the variadic *get_random_items()* function again:

```
def get_random_items(n, *choices):
    items = ()
    for i in range(n):
        index = random.randrange(len(choices))
        items = items + (choices[index],)
    return items
```

Let us call this function e. g. with 6 arguments to get 2 random elements. The 5 numbers would be packed in a tuple:

```
print(get_random_items(2, 1, 3, 5, 7, 9))    # e. g. (3, 9)
```

What about calling the function like this:

```
print(get_random_items(2, range(1, 10, 2)))    # (range(1, 10, 
2), range(1, 10, 2))
```

Watch the ouput in comment carefully.

Though the *range(1, 10, 2)* represents the same sequence of numbers as in previous call (numbers 1, 3, 5, 7, 9), the result is different. In fact, just one argument of the range type was passed into the function and so the *choices* parameter was set to reference a tuple of 1 element.

🕮 8.1.20

To **unpack elements from a structure** (before passing them into a variadic function or after getting a structured value from a function), we use the * operator.

Watch the outputs of printing out the values **unpacked from structures** (a range, a list and a string):

```
print(*range(10))                        # 1 2 3 4 5 6 7 8 9
print(*['mother', 'father', 'child'])    # 
mother  father  child
print(*str(2**30))                       # 1 0 7 3 7 4 1 8 2 4
```

As we have already seen many times, the *print()* function is also a nice example of a variadic function. All the values we need to print out are always packed in a tuple and passed into the function for printing.

🕮 8.1.21

So now, we call the *get_random_items()* function again, using the unpacking of values:

```
def get_random_items(n, *choices):
    items = ()
    for i in range(n):
        index = random.randrange(len(choices))
```

```
        items = items + (choices[index],)
    return items

print(get_random_items(2, *range(10)))
# e. q. (5, 2)
print(get_random_items(2, *['mother', 'father', 'child']))
# e. g. ('mother','mother')
print(get_random_items(2, *str(2**30)))
# e. g. ('1', '8')
```

8.1.22

We want to greet all friends saved in the *friends list*.

Complete the statement of calling the *welcome_everybody()* function correctly.

```
def welcome_everybody(shout, *persons):
    for p in persons:
        if shout:
            print('HALLO ', p, '!')
        else:
            print('Hallo ', p, '!')

friends = ['Julia', 'Kate', 'Simona', 'Natalie']
welcome_everybody(False, _____)
```

8.2 Tuples (programs)

8.2.1 Punctuation

There is a useful constant defined in the standard *string* module:

```
string.punctuation = r"""!"#$%&amp;-()*+,-
./:;<=&gt;?@[]^_`{|}~"""
```

Write the *get_punctuation()* function with 1 argument - a string to process.

The function returns a tuple containing all the punctuation characters found in the input string. The order of characters must be preserved.

```
Input : Hallo world!(1+1=2)
Output: ('!','(','+','=',')')
```

```
Input : To pay: 100$ + 10%; teddy@gmail.com
```

```
Output: (':','$','+','%',';','@','.')
```

```
Input : priscilla
Output: ()
```

8.2.2 All in One

Write the *all_in_one()* function that accepts variadic number of arguments - strings.

The function returns one string joining all the input strings together. The '*#*' serves as a delimiter.

```
calling the function:
print(all_in_one('mather','father','child','dog','cat'))
Output: mather#father#child#dog#cat
```

```
calling the function:
print(all_in_one('Slovakia'))
Output: Slovakia
```

8.2.3 Median and Modus

Write the *simple_statistics()* function having 2 arguments - an input list of *positive* integers and the maximum value that is saved in the input list.

The function returns *a tuple* with two elements - median and modus of the input number sequence

```
actual input parameters: [1,3,1,5,4] 5
Output: (3,1)
```

```
actual input parameters: [5,7,1,1,1,2,3,5,5,7,7,7,7,7] 7
Output: (5,7)
```

```
actual input parameters: [] None
Output: ()
```

8.2.4 Mission Completed

Imagine you are a captain driving a rocket in a 2D space that is represented by the Cartesian coordinate system.

The position of a rocket is specified by 2 coordinates (x and y) .Your mission always starts from center of the coordinate system (where x==0 and y==0).

A plan for a mission is coded with characters representing 4 cardinal points and integers representing number of steps to make in the appropriate direction. Consider the input list would contain following values (a mission-s plan):

```
mission_plan = ['N',3,'W',2,'S',1]
```

Your rocket will start at 0,0 and than move 3 times to the North (to the position 0,3). Next, the rocket will move 2 times to the West (to the position -2,3). Finally, you will move once to the South and so reach the final position -2,2.

Correct the presented solution to work properly, be careful about the following requirements:

The function must accept a variadic number of arguments (always even number, whatch the examples below).

The function must return *a tuple* containing rocket-s coordinates after finishing its actual mission as well as direct distance from its start position (2 decimals have to be printed in this case).

```
Input : N,3,W,2,S,1
Output: -2, 2, 2.83
```

```
Input : N,1,W,1,S,10,E,2
Output: 1, -9, 9.06
```

start.py

```
import math

# correct the following solution:

# plan must be a packed argument, not a list!
def go_on_mission(plan):
    x = 0
    y = 0
    for i in range(0, len(plan), 2):
        if plan[i] == 'N':
            y += plan([i + 1])
        elif plan[i] == 'S':
            y -= plan[i + 1]
        elif plan[i] == 'E':
            x += plan[i + 1]
        else:
            x -= plan[i + 1]
```

```
    d = math.sqrt(x * x + y * y)

# read a mission-s plan
data = input()

# call the function, unpack  actual argument values from the
data list
pos_x, pos_y, distance = go_on_mission()

# print out 3 elements of the returned tuple
print(f'{}, {}, {:.2f}')
```

Lists of Lists

Chapter 9

9.1 Lists of Lists

🕮 9.1.1

In lists or tuples, sequences of elements are saved. Lists are mutable, tuples are immutable.

The elements of lists may be of a different type, even a structured type.

Let us define a **list of tuple elements**:

```
a = (0, 1)
b = (1, 1)
c = (2, -5)
d = (3, 7)
e = (4, 8)
points = [a, b, c, d, e]
print(points)                # [(0, 1), (1, 1), (2, -5), (3, 7),
(4, 8)]
```

The *points* variable is a list containing 5 tuples. Each tuple represents a single point (its coordinates).

We can append new elements to the *points* list, update elements, delete them or use slicing:

```
points.append((5, 0))
points.pop(0)
print(points)                # [(1, 1), (2, -5), (3, 7), (4, 8),
(5, 0)]
points[2] = (1000, 1000)
print(points)                # [(5, 0), (4, 8), (1000, 1000),
(2, -5), (1, 1)]
print(len(points))
points[:] = points[::-1]
print(points)                # [(5, 0), (4, 8), (1000, 1000),
(2, -5), (1, 1)]
```

We can access the individual elements of a tuple using the second index:

```
print(points[0])             # (5, 0) - the first tuple in
a list
print(point[0][0])           # 5 - the first element of the
(5,0) tuple
print(point[0][1])           # 0 - the second element of the
(5,0) tuple
```

Of course, as tuple objects do not support item assignment, the following command would cause an error:

```
points[1][0] = 0            #... TypeError: 'tuple' object
does not support item assignment
```

9.1.2

What about **tuples of tuples**? Tuples can contain tuples as elements.

This may be useful in such situations when the number of elements as well as the elements themselves would not change over time.

```
points = (a, b, c, d, e)
print(points)                                   # ((0, 1),
(1, 1), (2, -5), (3, 7), (4, 8))
print(points[random.randrange(len(points))])    # e.g. (2, -
5)
```

To append a new element to the existing *points* tuple, a new tuple must be created:

```
points = points + ((1000, 1000),)
print(points)                       # ((0, 1), (1, 1), (2,-
5), (3, 7), (4, 8), (1000, 1000))
```

We joined the original *points* tuple with a tuple containing the new element. The *points* variable references a new tuple now.

9.1.3

A list of 3 visitors was created like this:

```
visitors = [('Susan', 45, 'Slovakia', True), ('Peter', 33,
'Germany', False)]
visitors.append(('Martin', 18, 'Hungary', False))
```

Match the following lines of code with their explanations:

```
print(visitors)
```

____

```
for v in visitors:
    print(v)
```

____

```
sum = 0
for v in visitors:
    sum += v[1]
print(f'average age: {sum/len(visitors)}')
```

____

```
for name, age, country, ok in visitors:
    print(f'{name} from  {country} is {age} years old.')
```

____

- Tuples in the visitors list were unpacked to 4 variables while iterating.
- The list of 3 tuples was printed out in one line.
- Ages of all visitors were sumed and the average was printed out.
- Each element in the visitors list was printed out separately.

🕮 9.1.4

We often need to implement a 2-dimensional structure – **a table with rows and columns** (a matrix of values from a mathematical point of view). Such table of values can be created using **lists of lists** in Python.

Let us create a list with 3 elements of the *list* type (a table with 3 rows and 3 columns):

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

Each of the *m[0]*, *m[1]*, *m[2]* elements are referencing a list having 3 elements (a row).

The same structure could be created also like this:

```
>>> row1 = [1, 2, 3]
>>> row2 = [4, 5, 6]
>>> row3 = [7, 8, 9]
>>> m =[row1, row2, row3]
>>> m
>>> [[1, 2, 3],[4, 5, 6], [7, 8, 9]]
```

Actual number of rows or columns can be determined using the *len()* function.

The length of the *m* list represents the **number of rows**:

```
print(len(m))       # 3
```

The length of the *m* list's element, e. g. the row *m[0]*, represents the **number of columns**:

```
print(len(m[0]))    # 3
```

The elements of the *m* structure are accessible via **a pair of indices**.

The *m[i][j]* represent a value of the element in row i and column j.

Let us change one of the elements and print the list of lists out:

```
m[0][0] = 0
print(m)            # [[0, 2, 3], [4, 5, 6], [7, 8, 9]]
```

9.1.5

Check all true statements about the *a* structure.

```
a = [[0, 0, 0, 0],[0, 0, 0, 0 ], [0, 0, 0, 0]]
```

- The a is a list of lists
- The a[1] is a list of int values.
- The a[2] references the last row of the a table.
- len(a) == len(a[0])

9.1.6

Consider the same *m* table as before:

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

To iterate through a table, we typically use the nested for cycles.

For every row, each of its elements is printed out:

```
for row in m:
    for x in row:
        print(x, end=' ')
    print()
```

The same output could be achieved in a shorter way, with help of the unpacking operator:

```
for row in m:
    print(*row)
```

The Output:

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

Sometimes, it is useful to process the elements of a table using their indices, e. g.:

```
x = 9
for i in range(len(m)):
    for j in range(len(m[0])):
        m[i][j] = x
        print(m[i][j], end=' ')
        x -= 1
    print()
```

The Output:

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

9.1.7

What is the output of following program?

```
def print_table(tab):
    for row in tab:
        print(*row)

t1 = ['a'] * 4
t2 = ['b'] * 4
t3 = ['c'] * 4
t = [t1, t2, t3]
t[0][2] = 'x'
print_table(t)
```

- [[a, a, x, a,], [b, b, b, b], [c, c, c, c]]
- [[a, a, a, a], [b, b, b, b], [x, c, c, c]]
- [[4a, 4a, x, 4a], [4b, 4b, 4b, 4a], [4b, 4c, 4c, 4c]]
- None of the options, the element with such indices does not exist.

🕮 9.1.8

When creating a list of lists, we must be very careful. Lists are mutable. And in lists of lists, **also their elements are mutable**.

Watch the result of creating the *m* table like this:

```
>>> m = [[0, 0, 0]] * 3
>>> m
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

There are 3 rows with 3 zeros.

Let us change one of the values:

```
>>> m[0][0] = -1
>>> m
[[-1, 0, 0], [-1, 0, 0], [-1, 0, 0]]
```

What happened? The elements *m[1][0]* and *m[2][0]* should not be changed.

In fact, with the assignment:

```
m = [[0, 0, 0]] * 3
```

we created a list with 3 references to the same list [0, 0, 0], just like this one:

```
>>> row = [0, 0, 0]
>>> m = [row, row, row]
```

We can also check the rows for identity easily, e. g.:

```
>>> m[0] is m[1]
True
```

🕮 9.1.9

To create a list of lists correctly, first, create an empty list and then, create and append each of the rows:

```
>>> m = []
>>> for i in range(3):
>>>     m.append([0,0,0])
```

Now, everything is ok:

```
>>> m[0][0] = -1
>>> m
[[-1, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> m[0] is m[1]
False
```

9.1.10

We need a table with 5 rows and 3 columns with all elements initialized to 0.

Which of the following options of creating it are correct?

a)

```
a = [ [0, 0, 0 ] for i in range(5) ]
```

b)

```
a = [[0, 0, 0]] * 5
```

c)

```
a = []
for i in range(5):
    a.append([0] * 3)
```

- a
- b
- c

9.1.11

For creating 2-dimensional structures, such as lists of lists, functions are very useful:

First, the *create_table()* takes the size of a table (number of rows and columns) and an element's default value.

The created list is returned.

```
def create_table(m, n, value=0):
    t = []
    for i in range(m):
        t.append([value] * n)
```

```
    return t

my_table = create_table(2, 3)
print(my_table)                        # [[0, 0, 0], [0,0,0]]
my_table = create_table(4, 2, '*')
print(my_table)                        # [['*', '*'], ['*',
'*'], ['*', '*'], ['*', '*']]
```

In function *create_table2()*, the *t* list with *m* elements having the *None* value is created at first.

The *t[i]* elements are prepared but not referencing rows yet.

Then, each of the reference is set to the corresponding row of *n* values within the *for* cycle.

```
def create_table2(m, n, value=0):
     t = [None] * m
     for i in range(m):
        t[i] = [value] * n
     return t

my_table = create_table2(3, 5, 1)
print(my_table)                      # [ [1, 1, 1, 1, 1], [1,
1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

9.1.12

Which of the two functions creates a copy of a table correctly?

```
def copy_table1(tab):
    t = []
    for row in tab:
        t.append(row)
    return t

def copy_table2(tab):
    t = []
    for row in tab:
       t.append(list(row))
    return t

m = [[1, 2, 3],[4, 5, 6], [7, 8, 9]]
m1 = copy_table1(m)
m2 = copy_table2(m)
```

- m1 is a copy of m
- m2 is a copy of m

9.1.13

A list of lists (or tuples) can be created also using list comprehension notation.

Watch some nice examples, the outputs are in comments:

```
t = [(i, j) for i in range(2) for j in range(3)]
print(t)
# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Notice, that each of the elements is a tuple. There are all possible combinations of *i* and *j*.

```
t = [[0] * 3 for i in range(4)]
print(t)
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Notice, that each of the rows is created separately.

```
t = [list(str(2 ** i)) for i in range(10)]
print(t)
# [['1'], ['2'], ['4'], ['8'], ['1', '6'], ['3', '2'], ['6',
'4'], ['1', '2', '8'], ['2', '5', '6'], ['5', '1', '2']]
```

Notice, that rows of the list are of different length as the result of 2 ** i is longer for higher exponent.

```
import random
t = [[random.randint(0, 9)] * 5 for i in range(3)]
print(t)
# e. g. [[8, 8, 8, 8, 8], [2, 2, 2, 2, 2], [7, 7, 7, 7, 7]]
```

Notice, that all elements in a row have the same random value.

9.1.14

What would be the program's output for the following list of input words?

To be, or not to be, that is the question.

```
words = input('Your words: ').split()
t = [list(w) for w in words if len(w) > 3]
print(t)
```

- ['that', 'question.']
- [['t', 'h', 'a', 't'], ['q', 'u', 'e', 's', 't', 'i', 'o', 'n', '.']]
- None of the options is correct, an error would occur.

📖 9.1.15

There are many situations, when we need a list of lists of various lengths.

For example:

- for numbers, we could have lists of their divisors,
- for graph vertices, we could have lists of their neighbours,
- for days of a week, we could have lists of incoming mails etc.

The following function takes *an iterable* containing the lengths of future rows as well as a default element's value:

```
def create_table(row_lengths, value=0):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [value] * row_lengths[i]
    return t
```

First, the *t* list containing *None* values is created. It has the necessary number of elements = future references to rows.

Next, all of the rows are created separately. Their lengths are taken form the first parameter.

Let us use the function *create_table()* in action:

```
t = create_table((1, 2, 3, 4, 5))
print(t)
# [[0],[0,0],[0,0,0],[0,0,0,0],[0,0,0,0,0]]

t = create_table(range(4), 'x'))
print(t)
# [[],['x'],[ 'x', 'x'], ['x', 'x', 'x']]
```

Notice, that *range(4)* represents a sequence of numbers 0, 1, 2, 3, so the very first row of *t* is an empty list.

9.1.16

Compare the following functions:

```
import random

def create_table1(row_lengths):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [random.randrange(2)] * row_lengths[i]
    return t

def create_table2(row_lengths):
    t = [None] * len(row_lengths)
    for i in range(len(t)):
        t[i] = [random.randrange(2) for i in
range(row_lengths[i])]
    return t
```

Match the function calls with correct outputs:

```
t = create_table1(range(1, 6))
print(t)
```

____

```
t = create_table2(range(1, 6))
print(t)
```

____

- [[0], [1, 1], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]
- [[1], [0, 0], [0, 0, 0], [1, 0, 0, 1], [0, 0, 0, 1, 0]]
- [None, None, None, None, None]

9.2 List of Lists (programs)

9.2.1 Matrix Initialization

The input integer number *n* represent the size of a square matrix.

Create a matrix with *n* rows and *n* columns and populate it with values *1..n * n*.

Display the initialized matrix as a list of lists in the output.

```
Input : 3
Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
Input : 2
Output: [[1, 2], [3, 4]]
```

9.2.2 O or X

The input integer number *n* represents the size of a square matrix *n*. The *n* is greater than 2.

Next, the individual rows of the input matrix are given.

The input matrix contains just values 0 and 1 and represents a black and white raster image (0 = black pixel, 1 = white pixel).

Write a program that detects whether a black letter O or a black letter X is stored in a matrix.

Output the correct characters 'O', 'X' (for letters O and X) or '?' in case it's something else.

```
Input : 3
0 0 0
0 1 0
0 0 0
Output: O
```

```
Input : 3
0 1 0
1 0 1
0 1 0
Output: X
```

```
Input : 3
```

```
1 0 0
0 0 1
1 0 0
Output: ?
```

⌨ 9.2.3 The Highest Peak

The m x n matrix contains numbers that represent altitudes of locations on a map.

Determine the coordinates (row and column) of the highest peek.

If there are more places on the map with the highest altitude, print out the northernmost. If there are more northernmost places with the highest altitude, print out the easternmost.

First, the dimensions of the matrix m, n are given followed by the individual rows of the matrix.

```
Input : 3 4
1 2 1 0
1 0 0 1
1 1 2 0
Output: 0 1
```

```
Input : 3 5
2 2 1 0 0
2 2 3 3 0
1 2 3 0 0
Output: 1 3
```

⌨ 9.2.4 Transforming Images

The input matrix with *m* rows and *n* columns contains information about a raster image of the corresponding size - the numeric color codes of the individual pixels.

Create a program that flips this image along the horizontal or vertical axis.

The input shows the dimensions of the matrix in the first row, followed by the individual rows of the matrix. The last line of the input contains the type of flip - by x-axis or y-axis.

```
Input : 3 4
1,1,1,1
0,1,0,1
1,0,0,1
```

```
x
Output: 1,0,0,1
0,1,0,1
1,1,1,1
```

```
Input : 3 4
1,1,1,1
0,1,0,1
1,0,0,1
y
Output: 1,1,1,1
1,0,1,0
1,0,0,1
```

```
Input : 1 5
1,2,3,4,5
y
Output: 5,4,3,2,1
```

9.2.5 Relationships between Persons

The input square matrix *a* with *n* rows and *n* columns contains information about mutual relations between *n* persons numbered 0, 1, ..., n-1.

If a [i] [j] == 1, it means that person *i* relates to person *j*.

If a [i] [j] == 0, it means that person *i* does not relate to person *j*.

People *i* and *j* are friends if and only if they relate to each other.

Create program to find out the person with highest number of friends.

First input number represents the number of persons (matrix size). The following lines are rows of the matrix.

If there are more people having the same number of friends, print out the person with a smaller number. Every person is self-related, but we will not count one as his/her own friend.

```
Input : 4
1 1 1 0
1 1 1 1
1 1 1 1
0 1 0 1
Output: person 1
```

```
Input : 3
1 1 0
0 1 1
1 0 1
Output: no friends
```

```
Input : 5
1 0 0 0 1
1 1 1 1 0
0 1 1 1 0
0 1 1 1 1
1 1 1 1 1
Output: person 3
```

9.2.6 Sorting Rows

There are data about charity funders (name, city, donation amount) saved in a textfile. The file is not empty, each line corresponds to one donor.

Save the input data in a table implemented as a list of lists.

The input consists of a file name and a request to sort the rows of table according to one of the columns (the sorting criterion is given as a column index).

Print out the table sorted by the specified criteria on the screen.

```
Input : textfile1.txt
0
Output: Alex,Bratislava,50
Boris,Zilina,10
Francis,Nitra,200
John,Nitra,100
```

```
Input : textfile1.txt
2
Output: Boris,Zilina,10
Alex,Bratislava,50
John,Nitra,100
Francis,Nitra,200
```

```
textfile1.txt:
John,Nitra,100
Francis,Nitra,200
Alex,Bratislava,50
```

```
Boris,Zilina,10
```

Sets

Chapter 10

10.1 Sets

🕮 10.1.1

Python contains a standard data type called *set* representing a set with features as we know them from math: elements are not repeated, their order does not matter.

In other words, **sets are unordered collections of unique elements**.

The elements of a set must be immutable, e. g. strings, numbers, tuples. It is not possible to have a set of lists, as lists are mutable structures.

We use different methods and set operations when working with sets:

Let us create **an empty set** and add some elements to it:

```
>>> m = set()
>> m
set()
>>> m.add('John')
>>> m.add('Paul')
>>> m.add('Francis')
>>> m
{'Paul', 'John', 'Francis'}
```

Three elements were added to the *m* set.

As seen in output, this collection does not save any information about the order of elements.

The set can be defined also by enumerating its elements using braces, e. g.:

```
>>> m = {'Theresa', 'Diana', 'Maria', 'Anna'}
>>> len(m)
4
```

The *len()* function returns the number of elements just like for any other iterable.

🕮 10.1.2

The *set()* constructor may take an optional argument – any iterable structure from which the set’s elements are taken:

```
>> set(range(3))
{0,1,2}
```

```
>> set('aeiouy')
{'a', 'u', 'i', 'o', 'e', 'y'}
 >>> type(m)
<class set>
```

By calling the *type()* function, the *set* type of the *m* variable can be verified easily.

Notice, that you **cannot create an empty set like this**:

```
>>> m = {}
>>> type(m)
<class 'dict'>
```

The *m* is **an empty dictionary**, not a set. We will work with dictionaries later.

10.1.3

Check all the correct definitions of sets:

- a = {1, 2, 3, 4, 5}
- b = set()
- c = {'Python'}
- e = {(1,2,3),(4, 5, 6)}
- f = set('Python')

10.1.4

What about trying to add the same value to a set repeatedly, e. g.:

```
>>> m = {'Theresa', 'Diana', 'Maria', 'Anna'}
>>> m.add('Maria')
>>> m
{'Theresa', 'Maria', 'Anna', 'Diana'}
```

As seen above, the string *'Maria'* was not added again as it is already present in the set.

A set can be updated with the union of itself and others using the *update()* method:

```
>>> m
{'Theresa', 'Maria', 'Anna', 'Diana'}
>>> m2 = {'John', 'Peter', 'Maria'}
>>> m.update(m2)
```

```
>>> m
{'Theresa', 'Maria', 'Anna', 'Diana', 'Peter', 'John'}
```

To determine, whether a set contains a specified element, the *in* operator is applicable:

```
>>> 'Anna' in m
True
>>> 'Regina' in m
False
```

10.1.5

Execute the following script and select the correct Output:

```
values = [1, 8, 3, 1, 2, 5, 3]

m1 = set()
m2 = set()

for v in values:
    if v in m1:
      m2.add(v)
    else:
      m1.add(v)

print(m2)
```

- {1, 3}
- {8, 2, 5}
- {1, 8, 3, 2, 5}

10.1.6

The elements can be removed from a set using its *discard()* method:

```
>>> m = set(range(10))
>>> m
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> m.discard(5)
>>> m
{0, 1, 2, 3, 4, 6, 7, 8, 9}
```

```
>>> m.discard(100)
>>> m
{0, 1, 2, 3, 4, 6, 7, 8, 9}
```

In case, the element is not in the set, nothing is done.

All elements can be removed from set at once calling the *clear()* method:

```
>>> m.clear()
>>> m
set()
```

To test a set for being empty, use one of these comparisons:

```
>> m == set()
True
>>> len(m) == 0
True
```

10.1.7

There are 2 other methods available for removing elements from sets:

The *pop()* method removes and returns an arbitrary set element and raises *KeyError* if the set is empty.

The *remove()* method removes an element from a set. If the element is not a member, raises a *KeyError*.

10.1.8

Execute the sequence of methods calls below.

Match the lines with corresponding explanations of the actual status:

```
girls = {'Theresa', 'Maria', 'Anna', 'Diana'}
```

____

```
girls.discard('Maria')
```

____

```
girls.pop()
```

____

```
girls.remove('Kate')
```

____

```
girls.clear()
```

____

```
girls.pop()
```

____

- The element ‘Maria’ was removed.
- The KeyError exception was raised.
- A set with 4 elements was created.
- An arbitrary element, e. g. ‘Diana’ was removed and returned.
- The girls set is empty.
- The KeyError exception was raised.

🕮 10.1.9

To realize the standard set operations easily, set operators are very useful:

```
>>> a = {1, 2, 3, 4}
>>> b = {4, 5}
>>> a | b
{1, 2, 3, 4, 5}
```

The | operator returns the **union** of sets as a new set (all elements that are in either set).

```
>>> a & b
{4}
```

The **&** operator returns the **intersection** of two sets as a new set (all elements that are in both sets)

```
>>> a - b
{1, 2, 3}
```

The **–** operator returns the **difference** of two sets as a new set (all elements that are in this set but not the other).

```
>>> a ^ b
{1, 2, 3, 5}
```

The **^** operator returns the **symmetric difference** of two sets as a new set (all elements that are in either set but not in both).

For determining the relations between sets, the relational operators are applicable:

```
>>> a = {1, 2, 3}
>>> b = {2, 3, 4, 5}
>>> c = set(range(2, 6))
>>> a == b
False
>>> b == c
True
```

The *a* and *b* are not equal as they do not contain same elements, but the *b* and *c* do.

```
>>> b < a
True
```

The *b* set is a **subset** of the *a* set (The *a* set contains all elements present in the *b* set).

The other operators (!=, <=, >, >=) can be used in a similar way.

If you do not like using the set operators, also methods are available for achieving the same goal (e. g. *union()*, *intersection()*, *issubset()* etc.). The advantage of using methods is that they accept any iterable structure as a parameter, e. g.:

```
a =  set('abcd')
b = 'cdefgh'
print(a.intersection(b)) # {'d', 'c'}
# print(a & b)   this is not applicable, operators work just
with sets
```

10.1.10

Match the operations on sets with correct outputs.

Notice, that before printing, we call the *sorted()* function in order to order the set elements (just to make the output more readable). The *sorted()* function takes an iterable and returns a list.

First, 2 sets of characters are created:

```
a = set('abc')
b = set('defg')
```

And then, the program continues:

```
c = a | b
print(sorted(c))    # _____
c = a & b
print(sorted(c))    # _____
c = c | {'x'}
print(sorted(c))    # _____
a |= {'d', 'e'}
print(sorted(a))    # _____

print({'a', 'b'} < a)   # _____
```

- True
- ['x']
- ['a', 'b', 'c', 'd', 'e', 'f', 'g']
- ['a', 'b', 'c', 'd', 'e']
- []

🕮 10.1.11

What about indexing or iterating through the elements of a set?

As sets are unordered collections, the **indexing takes no meaning** and is not applicable (an exception will occur):

```
>>> m = set(range(10))
>>> m[2]
Traceback (most recent call last):
  File "<input>", line 1, in <module> TypeError: 'set' object
does not support indexing
```

We can use *for* cycle to get all elements of a set sequentially, but **we cannot rely on any specific order while iterating**:

```
m = {5, 7, 9}
```

```
for e in m:
   print(e, end=' ')
```

One of possible outputs: 7 5 9

10.1.12

Study this short program using a set called *vowels*:

```
vowels = set('aeiouy')
print(len(vowels))
print(vowels)
```

Check all statements that are true for sure.

- The len(vowels) returns 6.
- The index of the last element is len(vowels)-1.
- The set's elements were printed out in same order as they are specified within the set() argument.

10.2 Sets (programs)

10.2.1 Simple Webmining

Users repeatedly log into a system from different IP addresses. We are interested in how many different IP addresses it was.

The input file contains in separate lines the IP addresses obtained from a server log file.

Use the set structure in your solution.

```
Input : logfile1.txt
Output: 7
```

```
logfile1.txt:
62.197.192.174
217.144.26.164
188.167.21.237
178.41.131.165
178.41.131.165
62.197.192.174
```

```
91.127.210.43
217.144.26.164
176.101.176.92
88.80.227.82
88.80.227.82
88.80.227.82
```

10.2.2 Simple Webmining II.

Users repeatedly log into a system from different IP addresses. Find out the number of IP addresses from which there was only one access to the system.

The input file contains in separate lines the IP addresses obtained from a server log file.

Use the set structure in your solution.

```
Input : logfile1.txt
Output: 3
```

```
logfile1.txt:
62.197.192.174
217.144.26.164
188.167.21.237
178.41.131.165
178.41.131.165
62.197.192.174
91.127.210.43
217.144.26.164
176.101.176.92
88.80.227.82
88.80.227.82
88.80.227.82
```

10.2.3 Union and Intersection

Print out the sets that represent the union and intersection of 2 non-empty input sets of integers.

Be careful and always print out each set's elements in ascending order!

```
Input : 1 2 3 4
1 2 5
Output: 1 2 3 4 5
```

```
1 2
```

```
Input : 1 2 3 4
5
Output: 1 2 3 4 5
empty set
```

10.2.4 Contest

The input set contains names of students who participated in the competition.

Remove all participants whose name begins with the specified string.

Print out a string containing names of all students who remained in the set.

Be careful and always print out each set's elements in ascending order!

```
Input : Peter Paul John Patrick Paula
Pa
Output: John Peter
```

```
Input : Amanda Ashly Aurel Barbara Betty Billy
xzy
Output: Amanda Ashly Barbara  Aurel Billy Betty
```

10.2.5 Brainstorming

Two teams were suggesting names for their software product. Their ideas were stored in input sets.

Print out all such ideas that were unique to the individual teams (no one from the other team did not propose the same name).

Be careful and always print out each set's elements in ascending order!

```
Input : Miranda Priscilla Penelope
Antonia Eugenia Felicia Miranda Octavia Penelope Fabia
Priscilla
Output: Antonia Eugenia Fabia Felicia Octavia
```

```
Input : Flip Flop Flap
Flip Flop Flap FantasticEd
Output: FantasticEd
```

⌨ 10.2.6 Vowels in a Word

Write a function that returns number of different vowels (a, e, i, o, u, y) contained in the input word.

Use the set structure in your solution.

```
Input : ukulele
Output: 2
```

```
Input : lalalala
Output: 1
```

⌨ 10.2.7 Cartesian Product

Write function *cartesian_product (a, b)*, which returns the Cartesian product of two sets containing integer numbers - set of all ordered pairs (tuples) whose first element is from the set *a* and the second component from the set *b*.

Be careful when printing the output. It must be a string with elements of the resulting set sorted in ascending order!

```
Input : 1 2 3
5 6
Output: (1, 5) (1, 6) (2, 5) (2, 6) (3, 5) (3, 6)
```

```
Input : 2
4 6 8 10
Output: (2, 4) (2, 6) (2, 8) (2, 10)
```

Dictionaries

Chapter 11

11.1 Dictionaries

🕮 11.1.1

Another useful data type built into *Python* is the **dictionary**.

Dictionaries are sometimes found in other languages as *associative arrays*. Unlike sequences, which are indexed by a range of numbers, *dictionaries are indexed by keys*, which can be any immutable type (typically strings or numbers).

It is best to think of a dictionary as ***an unordered set of key: value pairs***, with the requirement that ***keys are unique***.

A pair of braces creates an empty dictionary:

```
>>> d = {}
>>> d
{}
>>> type(d)
<class 'dict'>
>>> len(d)
0
```

The *dict()* constructor can be used to create an empy dictionary as well:

```
>>> dict()
{}
```

Remeber, {} is not an empty set.

Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary.

Let us create a translation dictionary (from English to Slovak), e. g.:

```
>>> d = {'mother': 'matka', 'father':'otec', 'house': 'dom',
'car': 'auto'}
>>> d
{'mother': 'matka', 'father': 'otec', 'house': 'dom', 'car':
'auto'}
```

📝 11.1.2

Match the examples of dictionaries with correct types of keys and values.

Example A

In this dictionary, words are mapped to their frequencies in a document:

```
a = {'and': 10, 'you': 12, 'meeting': 2 }
```

_____

Example B

In this dictionary, we associate sequences of test results with names of students:

```
b = {'Peter': [100,75, 99], 'Martin': [77,75, 77]}
```

_____

Example C

The ids of cards and their owners are saved in the dictionary as *key:value* pairs:

```
c = {728: 'Peter Carrot', 27: 'John Cabbage', 140: 'Andrea
Plum'}
```

_____

- key is a string, value is a list
- key is a string, value is an int
- key is an int, value is a string

🕮 11.1.3

The *dict()* constructor builds dictionaries also from sequences of *key:value* pairs:

```
my_list = [('mother', 'matka' ), ('father', 'otec'), ('house',
'dom'), ('car', 'auto')]
d = dict(my_list)
print(d)
# {'mother': 'matka', 'father': 'otec', 'house': 'dom', 'car':
'auto'}
```

In addition, *dict comprehensions* can be used to create dictionaries from arbitrary key and value expressions, e. g.:

```
d = {x: x**2 for x in (2, 4, 6)}
print(d)
# {2:4, 4:16, 6:36}
```

When keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
d = dict(Michael=77, Peter=66, Patrick=82)
print(d)
# {'Michael':77, 'Peter':66, 'Patrick':82}
```

11.1.4

Match the lines of code with resulting dictionaries.

```
>>> {x:random.randrange(x) for x in range(1,5)}
```

____

```
>>> dict(A=1, B=1.5, C=2, D=2.5, E=3, FX=4)
```

____

```
>>> dict([(2,'bin'),(8,'oct'), (16, 'hex')])
```

____

- {2: 'bin', 8: 'oct', 16: 'hex'}
- {'A': 1, 'B': 1.5, 'C': 2, 'D': 2.5, 'E': 3, 'FX': 4}
- {1: 0, 2: 0, 3: 2, 4: 3}

11.1.5

The main operations on dictionaries are

- storing a value with some key,
- extracting the value for a given key,
- deleting a *key:value* pair.

Let us work with a translation dictionary again:

```
>>> d = {'mother': 'matka', 'father':'otec'}
>>> d['child'] = 'dieta'
>>> d
```

```
{'mother': 'matka', 'father': 'otec', 'child':'dieta'}
```

With the assignement *d['child']= 'dieta'*, we added a new *key:value* pair to the dictionary.

If you store using a key that is already in use, the old value associated with that key is forgotten:

```
>>> d['child']= 'dieťa'
>>> d
{'mother': 'matka', 'father': 'otec', 'child':'dieťa'}
```

We access values using their keys:

```
>>> d['mother']
matka
```

It is an error to extract a value using a non-existent key:

```
>>> d['dog']
Traceback (most recent call last): File "<input>", line 1, in
<module>
KeyError: 'dog'
```

To check whether a single key is in dictionary, use the *in* keyword:

```
if 'dog' in d:
   print(d['dog'])
```

Or the *KeyError* might be useful:

```
word = input()
try:
   print(d[word])
except KeyError:
   print('Sorry, no such word in dictionary!')
```

11.1.6

The results of an exam were saved in this dictionary:

```
t = dict(A=10, B=12, C=8)
t['D']= 9
t['FX'] = 2
```

Match the following statements and expressions with correct results.

```
print(t)
```

____

```
t['C']
```

____

```
'FX' in t
```

____

```
t['X']
```

____

- { 'A':10, 'B':12, 'C':8, 'D':9, 'E':4, 'FX':2}
- True
- 45
- 8
- KeyError

11.1.7

Sometimes, we need to remove a *key:value* pair from a dictionary:

```
>>> d = {'mother': 'matka', 'father':'otec', 'car': 'auto'}
>>> d.pop('mother')
'matka'
>>> d
{'father': 'otec', 'car': 'auto'}
```

The *pop()* method removes the value associated with a given key and returns the value.

Also, the *del* operator is applicable:

```
>>> d = {'mother': 'matka', 'father':'otec', 'car': 'auto'}
>>> del d['mother']
>>> d
{'father': 'otec', 'car': 'auto'}
```

In both cases, the *KeyError* exception is raised if the given key does not exist.

For removing all pairs at once, the *clear()* method can be used:

```
>>> d.clear()
>>> d
{}
```

11.1.8

Select all true statements:

- The pop() method returns an arbitrary value from a dictionary.
- When using del, the KeyError exception might be raised.
- The clear() method makes the dictionary empty.

11.1.9

Sometimes, it is better to use the *get()* method for getting a value that is associated with a given key.

Compare the following situations:

```
>>> d = {'Slovakia': 'Bratislava', 'Austria': 'Vienna'}
>>> d['Slovakia']
'Bratislava'
>>> d.get('Slovakia')
'Bratislava'
```

The key is in the dictionary, so the corresponding value was returned.

```
>>> d = {'Slovakia': 'Bratislava', 'Austria':'Vienna'}
>>> d['Hungary']
Traceback (most recent call last): File "<input>", line 1, in
<module>
KeyError: 'Hungary'
>>> print(d.get('Hungary'))
None
```

The *get()* method did not raise a *KeyError*, the *None* value was returned.

When calling the *get()* method, we can specify a **default return value** as the second parameter (for the case of key non-existence):

```
>>> d.get('Hungary', '?')
'?'
```

11.1.10

Execute the following lines of code for 2 different user inputs (first 'C', then 'B').

Match the user inputs with the resulting status of the *t* dictionary.

```
t = { 'A': 10, 'C': 8, 'D': 9, 'E': 4, 'FX': 2}
x = input('next student's result:')
t[x] = t.get(x, 0) + 1
print(t)
```

```
x == 'C'
```

____

```
x == 'B'
```

____

- { 'A':10, 'B':0, 'C':8, 'D':9, 'E':4, 'FX':2}
- { 'A':10, 'B':1, 'C':8, 'D':9, 'E':4, 'FX':2}
- { 'A':10, 'C':9, 'D':9, 'E':4, 'FX':2}

11.1.11

Besides retrieving a value using its key, we can get **all keys**, **values** or **key:values pairs** (items) and iterate through them easily:

```
t = {'A': 10, 'B': 1, 'C': 8, 'D': 9, 'E': 4, 'FX': 2}
print(t.keys())
print(t.values())
print(t.items())
```

The Output:

```
dict_keys(['A', 'B', 'C', 'D', 'E', 'FX'])
dict_values([10, 1, 8, 9, 4, 2])
dict_items([('A', 10), ('B', 1), ('C', 8), ('D', 9), ('E', 4),
('FX', 2)])
```

All the returned objects seen above are **iterable**:

```
for k in t.keys():
    print(k, end=' ')       # A B C D E FX
```

```
for v in t.values():
    print(v, end=' ')       # 10 1 8 9 4 2

for k, v in t.items():
    print(k, v)
```

The Output:

```
A 10
B 1
C 8
D 9
E 4
FX 2
```

The same output as above could be produced also like this:

```
for k in t:
    print(k, t[k])
```

The returned values of calling the *keys()*, *values()* or *items()* can be converted to a list.

Performing *list(d.keys())* on a dictionary *d* returns a list of all the keys used in the dictionary, **in arbitrary order**.

In case we need it sorted, just use *sorted(d.keys())* instead.

11.1.12

Consider a dictionary in which key words and number of their occurrences in a source file are saved:

```
d = {'and':10, 'for':2, 'if':8}
```

Complete the *for* cycle in order to output all pairs from dictionary on a console:

```
for  k, v in _____:
    print(k, v)
```

11.1.13

A dictionary can be **an element of a list** as well as **a value in another dictionary**, e. g.:

```
s1 = {'name': 'Paul Carrot', 'address': {'Street': 'Long',
'number': 13,'city': 'Nitra'} }
s2 = {'name': 'Peter Pier', 'address': {'Street': 'Short',
'number': 21, 'city': 'Nitra'} }
s3 = {'name': 'Patrick Nut', 'address': {'Street': 'Deep',
'number': 77, 'city': 'Nitra'} }
students = [s1, s2, s3]
```

We created a list with 3 elements – *dictionaries*.

The *students[0]* element is a dictionary containing 2 *key:value* pairs:

```
>>> students[0]['name']
Paul Carrot
>>> students[0] ['address']
{'street': 'Long', 'number': 13,  'city': 'Nitra'}
```

The value associated with *'address'* key is a dictionary with its own keys:

```
>>> students[0]['address']['street']
Long
>>> students[0]['address']['number']
13
>>> students[0]['address']['city']
Nitra
```

11.1.14

Print out the year of a person' birth:

```
person = {'name': 'Mathew', 'birthday': {'day': 12,'month':
3,'year': 2000}}
print(person['birthday']_____)
```

11.1.15

In Python, *keyword arguments* are used very often when calling functions:

```
>>> a = [720, 1006, 2001, 77, 1555, 5002]
>>> sorted(a, key = lambda x: x % 10, reverse = True)
[77, 1006, 1555, 5002, 2001, 720]
```

The *a* list was sorted according to last digits of the elements, in descending order (parameters named *key* and *reverse* were used).

When using keyword arguments, we are not required to specify them in a function call (they are optional). In such cases their default values are used, e. g.:

```
>>> sorted(a, key = lambda x: x % 10)
[720, 2001, 5002, 1555, 1006, 77]
```

Default value for the *reverse* argument is *False*.

We also do not need to remember any specific order of keywords arguments for functions as they are identified with keywords, e. g.:

```
>>> sorted(a, reverse = True, key = lambda x: x % 10)
[77, 1006, 1555, 5002, 2001, 720]
```

The keyword arguments were written in another order, but output is the same.

🕮 11.1.16

What about **writing functions having keywords arguments**?

If there is a specific number of them, we just enumerate them using *pairs of keywords and default values* (after the last positional argument with no default value), e. g.:

```
def my_print(item, price=0, available=True):
    print(f'The {item} costs {price} Eur ', end ='')
    if available:
        print('and is available.')
    else:
        print('but is not available.')
```

Now, we call the function in various ways, watch the commented outputs:

```
my_print('T-Shirt')
# The T-Shirt costs 0 Eur and is available.

my_print('Calendar', price=1.5, available=False)
# The Calendar costs 1.5 Eur but is not available.

my_print('Hat', price=12)
# The Hat costs 12 Eur and is available.
```

📖 11.1.17

But what if we do not know the number of keyword arguments in advance or if there are too many of them?

This situation can be solved easily using the **keyword argument packing**:

Notice, the ** before the *features* parameter:

```
def my_print(item, **features):
    print(f'{item}:')
    for k, v in features.items():
        print(k, v)
```

The *features* parameter is a dictionary. We can call its *items()* method and print out the pairs of keys and values.

Be careful and do not rely on any specific order as **dictionaries are unordered collections**.

Now, we can call the function *my_print()* with arbitrary number of *keyword=value* pairs. They will be packed into a dictionary.

```
my_print('T-Shirt', color='red', price=10, available=False,
size='XL')
```

The Output:

```
T-Shirt:
color red
price 10
available False
size XL
```

The ** operator serves also for unpacking the keyword arguments from a dictionary within a function call:

```
d = {'color':'blue','price':10,'size':'M'}
my_print('T-Shirt',**d)
```

The Output:

```
T-Shirt:
color blue
price 10
size M
```

11.1.18

What would be the output of calling this function?

```
def f(**d):
    print(type(d), end='')
    print(d)

f(name='Peter', age=12)
```

- <|class 'dict'> {'name': 'Peter', 'age': 12}
- <|class 'dict'> {name: 'Peter', age: 12}
- <|class 'list'> [('name', 'Peter'), ('age', 12)]

11.2 Dictionaries (programs)

11.2.1 Frequency Analysis

Analyze text that is stored in the input string. Use the dictionary structure effectively.

Print out:

- number of different characters found in the text
- sequence of those different characters arranged in ascending order
- number of different characters that appear more than once in the text
- number of occurrences of the specified character

First, the text string is given, Next, one character whose frequency we are interested in follows.

```
Input : aaabccccdeee
a
Output: 5
a,b,c,d,e
3
3
```

```
Input : xyz
p
Output: 3
x,y,z
```

```
0
0
```

11.2.2 Translator

Our program already contains a dictionary for easy translation of words from English to Slovak. Do not change its content!

Finish the program like this:

Read and translate the input sentence. The sentence is defined as a sequence of English words written in lowercase letters. The word separator is always a space.

If any of the English words is not in the dictionary, print it in uppercase.

```
Input : your hat is big
Output: tvoj klobuk je velky
```

```
Input : my dog is happy
Output: moj pes je stastny
```

```
Input : his cat is nice
Output: HIS CAT je pekny
```

file1.py

```
# do not modify this line:

d = {'my': 'moj', 'dog': 'pes', 'hat': 'klobuk', 'is': 'je',
'nice': 'pekny', 'your': 'tvoj', 'happy': 'stastny',
     'father': 'otec', 'boy':'chlapec', 'big':'velky', 'and':
'a'}

# write your code here:
```

11.2.3 Chemical Calculator

Complete the program to calculate mass of a chemical compound given by its chemical formula.

Chemical element weights are saved in a text file named *elements.csv* (we are interested in the chemical element symbol, which is the second piece of data and the molecular weight of its atom, which is the last piece of data within the same line):

```
elements.txt:
Hydrogen,1,H,1.01
Helium,2,He,4.00
Lithium,3,Li,6.94
Beryllium,4,Be,9.01
Boron,5,B,10.81
Carbon,6,C,12.01
Nitrogen,7,N,14.01
Oxygen,8,O,16.00
...
```

Use dictionary to store masses of the chemical elements.

Read the formula of a compound from the input and print out the molecular mass of its molecule.

For simplicity, we use the dot symbol as a separator in a formula notation, for example:

H2SO4 (sulfuric acid) is written as **H2.S.O4**

To calculate the mass of sulfuric acid, we must add 2x the mass of the hydrogen atom (H), 1x the mass of the sulfur atom (S) and 4x the mass of the oxygen atom (O).

If the formula notation cotains an element symbol that does not exist (it is not in our table), print out an error message.

```
Input : H2.O
Output: 18.02
```

```
Input : C.O2
Output: 44.01
```

```
Input : Hi3
Output: error
```

```
Input : H2.S.O4
Output: 98.08
```

start.py

```
# write your code here (complete the follwing program
correctly):

def read_input_file():
    d = {}
```

```
        # process the input text file named 'elements.csv'
    return d

def count_molecule_mass(formula):
    result = 0.0
        # process the formula given as a string (e. g. 'H2.O' ,
'H2.S.O4' etc.)
    return round(result, 2)

# main program
table = read_input_file()
formula = input()
print(count_molecule_mass(formula))
```

11.2.4 Morse Code

A dictionary *table* is automatically prepared and filled with data from the input file *morse.csv*.

In this dictionary, Morse codes are associated with corresponding capital letters of the English alphabet and digits. Letters or digits are the *keys*, sequences of dots and commas represents the *values*.

Program the *get_message ()* function with 2 parameters: a dictionary mentioned above and a message written in the Morse code. This function must result in decoded message.

In the *morse* input parameter, we use spaces as delimeters of codes. You can assume, that all messages are one-word.

```
Input : ... --- ...
Output: SOS
```

```
Input : . ..- .-. --- .--. .
Output: EUROPE
```

```
morse.csv:
A,.-
B,-...
C,-.-.
D,-..
E,.
F,..-.
G,--.
H,....
```

```
I,..
J,.---
K,-.-
L,.-..
M,--
N,-.
O,---
P,.--.
Q,--.-
R,.-.
S,...
T,-
U,..-
V,...-
W,.--
X,-..-
Y,-.--
Z,--..
0,-----
1,.----
2,..---
3,...--
4,....-
5,.....
6,-....
7,--...
8,---..
9,----.
```

11.2.5 Access Codes

Employees in a company have got their identification numbers (positive integers).

In order to permit access to a specialized laboratory to a person, an access code must be assigned to him/her.

In the input file, pairs of data are stored on each line: employee identification number and his/her access code. Save this data in a dictionary.

Next, read an identification number from the input and cancel access to the lab for all employees whose identification number is equal or greater than or to the specified one.(remove all relevant key-value pairs from dictionary).

After updating the dictionary, print out the largest access code that is present in the dictionary (is assigned to an employee).

If the dictionary is empty after deleting elements, print out an 'error' message.

```
Input : 101
Output: 6955
```

```
Input : 10
Output: error
```

```
Input : 201
Output: 8541
```

```
access_codes.csv
101,5303
10,6955
45,6810
126,8541
201,5881
261,5753
357,2277
156,6559
530,9979
203,8698
```

11.2.6 Divisors

For a given natural number *n*, generate a dictionary for saving all divisions of all numbers from interval <1, n>.

Output the entire dictionary. Divisiors are in ascending order.

```
Input : 4
Output: {1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 4]}
```

```
Input : 7
Output: {1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 4], 5: [1,
5], 6: [1, 2, 3, 6], 7: [1, 7]}
```

priscilla.fitped.eu