

# Python – Data Structures

Ján Skalka  
Ľubomír Benko  
Vaida Masiulionytė-Dagienė  
Peter Švec  
Júlia Tomanová

[www.fitped.eu](http://www.fitped.eu)

2024

# Python – Data Structures

## Published on

*November 2024*

## Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Vaida Masiulionytė-Dagienė | Vilnius University, Lithuania

Peter Švec | Teacher.sk, Slovakia

Júlia Tomanová | Constantine the Philosopher University in Nitra, Slovakia

## Reviewers

Piet Kommers | Helix5, Netherland

Roman Valovič | Mendel University in Brno, Czech Republic

Cyril Klimeš | Mendel University in Brno, Czech Republic

Vladimiras Dolgopolas | Vilnius University, Lithuania

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095



**Funded by  
the European Union**

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.



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.

© 2024 Constantine the Philosopher University in Nitra

**ISBN 978-80-558-2226-6**

# TABLE OF CONTENTS

1 Sandbox.....	6
2 Exceptions.....	8
2.1 Avoiding errors.....	9
2.2 Exception processing.....	12
2.3 Different types of errors.....	17
2.4 Exceptions generating.....	25
2.5 Exceptions (programs).....	31
3 Files.....	33
3.1 Files.....	34
3.2 Reading and writing.....	38
3.3 More operations.....	44
3.4 Files (examples and programs).....	49
4 Lists.....	59
4.1 List.....	60
4.2 List elements.....	63
4.3 List editing.....	68
4.4 List (programs).....	74
5 List Processing.....	78
5.1 Creating a list.....	79
5.2 Processing elements.....	84
5.3 Slices.....	90
5.4 List comprehension.....	94
5.5 List (Programs II.).....	99
6 Lists and Memory.....	102
6.1 List in memory.....	103
6.2 List elements.....	108
6.3 Arrangement.....	114
6.4 List as a parameter.....	118
6.5 Programs.....	125
7 Tuple.....	129
7.1 Tuple.....	130
7.2 Elements manipulation.....	134
7.3 Use of tuple.....	139
7.4 Variadic function.....	142
7.5 Programs.....	146

8 List of Lists.....	152
8.1 Matrix introduction .....	153
8.2 Tables .....	158
8.3 Operations in a table .....	162
8.4 Reading and processing data .....	168
8.5 Data in files.....	178
8.6 Matrix in math .....	185
8.7 Matrix (programs).....	189
8.8 Tables (programs).....	191
9 Set .....	197
9.1 What is set.....	198
9.2 Comparison.....	202
9.3 Working with set .....	204
9.4 Programs (set).....	208
10 Dictionary .....	213
10.1 What is dictionary .....	214
10.2 Dictionary iterating .....	219
10.3 Typical examples.....	223
10.4 Programs (dictionary) .....	229

# Sandbox

## Chapter **1**

In its online version, this course is designed in such a way that in almost every place there is a window in which you can write and run any Python code.

# Exceptions

Chapter **2**



## 2.1 Avoiding errors

### 2.1.1

#### Exception

Exception, or an exceptional state (exception) is a situation in the program that occurs in the case of an error **during the program's execution** and usually interrupts/terminates the execution of the program.

Many errors can be predicted and treated directly by the program or programmed so that they will not occur.

A typical example is avoiding division by zero:

```
a = 1
b = 0
if b == 0:
    print("cannot be divided by zero");
else:
    print("share:", a / b);
```

In the code, we check whether the variable b does not contain a value that could cause the program to crash, and if so, we do not perform an unsafe operation, but inform the user that the operation cannot be performed.

This way we prevent an operation that could cause an error.

### 2.1.2

Arrange the function and code lines so that the program correctly handles division by zero.

- def calculation(a, b):
- result = calculation(10, 0)
- 
- return "error"
- print(result)
- if b != 0:
- else:
- return a / b

### 2.1.3

However, there are often situations in which programming the treatment of a potential error is too complicated, or several types of errors can occur on one line.

A typical operation is e.g. loading inputs and their subsequent conversion to a number.

When a non-numeric value is entered, the following code causes an error:

```
ta = input()
a = int(ta)
print(a)
```

**Program output:**

```
tenValueError
invalid literal for int() with base 10: 'ten'
```

Checking such a situation is quite simple:

```
value = input("Enter a number: ")
if value.isdigit():
    number = int(value)
else:
    print("conversion error")
    number = 0
```

**Program output:**

```
Enter a number: twoconversion error
```

## 2.1.4

Complete the code so that the loading of a positive integer value repeats until the user enters a valid value:

```
while ____:
    input_t = input("Enter a positive integer: ")
    if input_t.____():
        number = int(input_t)
    ____
else:
    print("Not a positive integer. Again.")
```

## 2.1.5

Checking the correctness of the whole number is a bit more complicated, while negative numbers can also be entered.

In such a case, the use of the `isdigit()` function is no longer sufficient for us, because the sign "-" is not considered a digit.

We can divide the verification of correctness into several conditions, while we must treat single-digit numbers separately:

```

value = input()
if (len(value) == 1 and value[0] in '123456789'): # if it is a
single-digit number
    number = int(input_t)
elif value[0] in '-123456789': # valid beginning of a number
    if (value[1:].isdigit()): # checking remaining values
        number = int(value)
    else:
        print("invalid input")
else:
    print("invalid input")

```

### 2.1.6

For which expressions does the `isdigit()` function return False?

- 42.75
- abc
- 100,000
- -538
- 0
- 007
- 12345
- 98765 43210

### 2.1.7

#### Replace()

To solve many tasks, Python programmers use the so-called sparrow cannon.

They can simplify the assigned task with a programming-complicated operation, which can be written in one command in Python.

An example can be verifying the correctness of an input of a real number, where they replace the first occurrence of a dot with an empty string and then check whether the remaining characters are numbers:

```

value = input("Enter a number: ")

# Checking whether the input without the first occurrence of a
dot is a numeric string
if value.replace(".", "", 1).isdigit():
    number = float(value)
    print(f"You entered a valid number: {number}")

```

```
else:
    print("This is not a valid decimal number.")
```

**Program output:**

```
Enter a number: 12.15You entered a valid number: 12.15
```

The **replace()** function replaces the occurrence of the character "." for an empty string, but it is limited for only the first occurrence (third parameter).

**📄 2.1.8 Decimal number input**

Write a program to check whether the input is a correct real number. If yes, separate its whole and decimal parts with a dash, if not, write "incorrect data". Do not use try - except and do not consider the scientific format of a number,

```
input: 3.14
output: 3 - 14
```

```
input: -42.75
output: -42 - 75
```

```
input: 12.34.56
output: incorrect data
```

**2.2 Exception processing****📄 2.2.1**

The codes providing the processing of inputs presented in the previous tasks are quite extensive and it is inefficient to devote a significant part of the code to their processing. In addition, it is likely that the programmer cannot predict all situations that may occur anyway.

In all modern programming languages, there is a tool through which we can create blocks in the code that allow us to catch an exception and deal with it.

In Python, it is try - except in the following form:

```
# some introductory code
try:
    # commands during which an error can occur
except:
    # what to do if an error occurred
# code continues
```

The beginning of the **try** block monitors whether an exception occurs. If it occurs, it does not continue with the commands, but jumps to the **except** block and executes the commands in it (usually error information).

Unless an error occurs between the **try-except** statements, the program skips the statements in the **except** block and continues beyond it.

### 2.2.2

In Python, which pair of statements ensures that an exception is caught and handled?

- try - except
- try - expect
- try - catch
- try - stop

### 2.2.3

A program to catch the exception for converting the input to a number could then look like this:

```
ta = input()
try:
    a = int(ta)
    print(a)
except:
    print('an error in conversion occurred')
print('code proceeds')
```

If an error occurs with the commands in the 3rd or 4th line the **except** block takes control and executes the commands specified in it. In this case, it's the printout of an error. After the command in the **except** block is executed, it continues beyond it.

If an error occurs during a command in the **try** section, the commands that are listed after the line with the error are not executed - it immediately proceeds to the **except** block.

If you enter a non-numeric input, the **print** command in the 4th line will no longer be executed.

### 2.2.4

Complete the program for loading a decimal number:

```
ta = input()
_____:
```

```

a = _____(ta)
print(a)
_____
:
print('conversion error occurred')
print('the code proceeds')

```

- try
- try
- int
- float
- except
- str
- double
- expect

## 2.2.5

### Try extend

How big the try block should be can be discussed, its extend can be different for different tasks, there are also situations when it makes sense to insert practically the entire code into it.

If, for the execution of the program, we need a variable to be available, the value of which is checked and assigned in the try block, it is necessary to fill it with some initial value before assigning the input to it, e.g.:

```

ta = input()
g = -1
try:
    g = int(ta)
except:
    print('conversion error occurred')
print('the code proceeds')
print(g)

```

#### Program output:

```

tenconversion error occurred
the code proceeds
-1

```

If we were to limit its existence to try, and an error would occur, the variable would not be defined:

```

ta = input()
try:
    h = int(ta)

```

```

except:
    print('an error after the conversion occurred')
print('code proceeds')
print(h)

```

**Program output:**

```

tenan error after the conversion occurred
code proceeds
NameError
name 'h' is not defined

```

An alternative is to set the default value in the **except** section:

```

ta = input()
try:
    i = int(ta)
except:
    print('an error after the conversion occurred')
    i = -1
print('the code proceeds')
print(i)

```

**Program output:**

```

tenan error after the conversion occurred
the code proceeds
-1

```

## 2.2.6

Complete the program for processing numerical inputs for division:

```

ta = input()
_____:
    a = int(ta)
_____:
    print('an error during the 1st number conversion occurred')
    a = 1 # we set some default value so that we don't have to
end the program
tb = input()
_____:
    b = int(tb)
_____:
    print('nan error during the 2nd number conversion occurred')
    b = 1 # we set some default value so that we don't have to
end the program
_____:

```

```

c = a / b
print("the share is", _____)
_____ :
print('a division error occurred')

```

- try
- c
- catch
- expect
- try
- try
- expect
- try
- except
- except
- a
- except
- b
- expect
- try

### 2.2.7

A common standard is to use a function to handle inputs. It should behave as we need and can interrupt code execution and continue from where it was called.

An effective use of the function to retrieve an integer can take the form:

```

def secondSquare(ta):
    try:
        a = int(ta)
        return a * a
    except:
        return 'error' # or it returns -1 as an undefined value

x = secondSquare(input())
if x == 'error':
    print('It is pointless to continue.')
else:
    print('I have my x =', x)

```

### 2.2.8

Complete the code to calculate the share of two numbers. If division by zero occurs, return -1:



```

def share(numerator, denominator):
    if denominator == 0:
        return -1
    else:
        share = numerator / denominator
        return share

result = share(20, 0)

if result == -1:
    print("Division by zero.")
else:
    print(f"The share is {result}")

```

## 2.3 Different types of errors

### 2.3.1

#### Various types of errors

In programming, it is not uncommon for a several different errors to occur on several lines. The program must inform the user what error has occurred and execute different code for each error.

For this purpose, we use `except`, for which we indicate the error for which its code is intended. In order to be able to correctly describe what kind of exception it is, we need to know the types of errors.

A typical example is code that combines the conversion problem and division by zero:

```

try:
    a = int(input())
    b = int(input())
    print(a // b, a % b)
    print('done')
except ZeroDivisionError:
    print('we do not divide by zero')
except ValueError:
    print('a non-numerical value was entered')

```

- the **ZeroDivisionError** exception occurs in case of division by zero

- a **ValueError** exception occurs if the input value cannot be converted to an integer of type int

Depending on the type of exception, only the part of the code in **except** that is intended for it will be executed.

### 2.3.2

Complete the correct definition of an exception:

```
try:
    a = int(input())
    b = int(input())
    print(a // b, a % b)
    print('done')
except ____:
    print('we do not divide by zero')
except ____:
    print('a non-numeric value was entered')
```

- ZeroDivisionError
- ConversionError
- ValueError
- ZeroError
- DefaultError
- UniversalError

### 2.3.3

The most common types of exceptions make it possible to estimate why and what kind of error occurred:

**ValueError** - an error occurred while converting or processing the value, e.g.:

- string to number conversion
- using a function that expects a certain type of data and receives another
- attempt to access an index within a string that does not exist
- other unspecified situations

**TypeError** - an error occurred during an operation for which the combination of data types is not valid, e.g.:

- mathematical operation between incompatible data types (number and text addition)
- using operations that are defined only for certain data types (trying to call a function that is not defined for a given object type)
- incorrect use of arguments when calling a function

- generally unfit combination of data types

**NameError** - an error occurred due to a used variable or function name that is not defined or valid, e.g.:

- using a variable or function name that has not been previously defined
- attempt to access variables that are not visible in the current scope
- mistake in the name

**ZeroDivisionError** - division by zero

**IndexError** - an error occurred while trying to access an index or list item that does not exist, eg:

- accessing an index that is larger or smaller than the range of valid indexes
- access index in empty list (strings)

**UnboundLocalError** - an error occurred while trying to access a variable that has not yet been defined but "soon" will be, e.g.:

- using a variable in a function before it is assigned a value

```
def mine():
    print(x)
    x = 100 ...
```

**FileNotFoundError** - an error occurred while trying to open or work with a file that does not exist or is not in the specified path

Examples of error printout with details:

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

## 2.3.4

What exception errors can be expected in the following situations?

A program that loads a string from the user, tries to convert it to an integer and prints the result. If the user enters a string that is not a valid number (such as "ABC"), \_\_\_\_\_ is generated.

A program that allows the user to enter two values and tries to sum them. If the user enters values of different data types (such as string and number), expect a \_\_\_\_\_.

A program with a function that uses a variable that is not defined within the function but should be. Attempting to access this variable before defining it should throw a \_\_\_\_\_.

A program that allows the user to enter two numbers and tries to divide one number by zero. If the user enters zero as the denominator, expect a \_\_\_\_\_.

A program that works with a list and tries to access items that are not within the range of valid indexes of the list. If the user specifies an out-of-range index, expect an \_\_\_\_\_.

A program with a function that attempts to use a variable that is defined only in another scope (such as the global context) but is not defined within the function. This can cause an \_\_\_\_\_.

A program that attempts to open a file for reading, using a path to a file that does not exist. Expect \_\_\_\_\_.

- FileNotFoundError
- UnboundLocalError
- ZeroDivisionError
- NameError
- TypeError
- ValueError
- IndexError

### 2.3.5

What exception errors can be expected in the following situations?

A program with a function that attempts to use a variable that was not previously defined within the function. This should throw a \_\_\_\_\_.

A program that allows the user to enter two numbers and tries to divide them. If the user enters zero as the denominator, expect a \_\_\_\_\_.

Create a program that allows the user to enter their name and age. The program will try to match the string with the number (age). Expect \_\_\_\_\_.

A program that prompts the user for an index and tries to retrieve an item from the list. If the user enters a bad index, the program should throw an \_\_\_\_\_.

A program that has a variable name error. Expect \_\_\_\_\_ when attempting to access a non-existent variable.

A program that allows a user to enter their age. If the user enters a string that is not a valid number, expect a \_\_\_\_\_.

A program that calculates the average value from a list of numbers. If the list is empty, the program should throw a \_\_\_\_\_.

A program that allows the user to enter two values and tries to multiply them. If the user enters values of non-integer data types, expect \_\_\_\_\_.

- ValueError
- NameError
- TypeError
- UnboundLocalError
- TypeError
- NameError
- ZeroDivisionError
- ValueError
- UnboundLocalError
- ZeroDivisionError
- IndexError
- NameError
- IndexError
- ValueError
- NameError
- TypeError
- FileNotFoundError
- TypeError
- FileNotFoundError

### 2.3.6

#### General exception

As we could see, there are quite a few exceptions, and the ones we mentioned certainly do not represent a complete list. The goal of the programmer is to treat expected exceptions in a special way, but also to ensure that the program completes its activity even if there is an exception that we specifically did not anticipate occurs.

If in our program from the previous task we try, e.g. access a non-existent index, program execution is aborted:

```

try:
    aa = input()
    a = int(aa)
    b = int(input())
    print(a // b, a % b)
    print("The second digit of the first number is",aa[1])
    print('done')
except ZeroDivisionError:
    print('we do not divide by zero')
except ValueError:
    print('a non-numeric value has been entered')
print("program continues...")

```

**Program output:**

```

5 60 5
IndexError
string index out of range

```

In the example, an exception of a type other than those listed occurred, it was not handled, and program execution ended.

Thus we will complete the block intended for processing all other exceptions - which are not listed:

```

try:
    aa = input()
    a = int(aa)
    b = int(input())
    print(a // b, a % b)
    print("The second number of the first number is",aa[1])
    print('done')
except ZeroDivisionError:
    print('we do not divide by zero')
except ValueError:
    print('a non-numeric value has been entered')
except:
    print('different exception')
print('program continues...')

```

**Program output:**

```

5 60 5
different exception
program continues...

```

 **2.3.7**

Complete the program so that it catches each type of exception:

```

_____ :
    a = int(input())
    b = random.randrange(-10,10)
    print(a, b, a // b)
    print("The second digit of the first number is",str(a)[1])
    print('done')
_____ :
    print('an error occurred')
print('all proceeds...')

```

- except ValueError
- except ZeroDivisionError
- try
- except UniversalError
- try-all
- except

### 2.3.8

#### Exception detail

Even if we use the general exception, we will not lose detailed information about it. Each exception that occurs fills a variable with information about the situation, which we can access through the following notation:

```

try:
    # some code causing an exception
except Exception as err:
    print(err) # printout of an information about an error

```

An example of use can be a situation generating already known problems:

```

try:
    a = int(input())
    b = int(input())
    print(a // b, a % b)
except Exception as err:
    print('error:', err) # the err object can be
    printed
    print('type:', type(err)) # we can find its type
    print('details:', err.args) # details about an error

```

#### Program output:

```

aa: invalid literal for int() with base 10: 'aa'
type:
details: ("invalid literal for int() with base 10: 'aa'",)

```

In the exception(**except**) processing entry, we used the Exception type, which represents a general exception. It is the type from which all other exceptions are derived and thus provides a universal type, the details of which can be obtained by inserting it into the err variable by writing as err:

- err - contains all information about the error
- type(err) - returns an error type, e.g. **ValueError**
- err.args - displays details about an error

### 2.3.9

Fill in the correct parameters/commands for the error statement:

```
try:
    a = int(input())
    b = int(input())
    print(a // b, a % b)
except Exception as err:
    print(_____)      # prints: ('integer division or modulo by
zero',)
    print(_____)      # prints: integer division or modulo by
zero
    print(_____)      # prints: <class 'ZeroDivisionError'>
```

- type(err)
- err.args
- err.type()
- err.detail
- err.args()
- err
- err.detail()

### 2.3.10

We can catch the situation when no exception occurs in the code with two special blocks:

- else
- finally

```
try:
    # commands
except Exception1:
    print('input has to be integer values')
except Exception2:
    print('zero division')
else:
```



```
print('all is ok')
```

The **finally** section has a special status, it is intended for commands that are to be executed regardless of whether or not an exception occurred in the try block. It is used e.g. to close the file if an error occurred/did not occur while working with it. However, it is not logically necessary...

```
try:
    # commands
except Exception1:
    print('input has to be integer values')
except Exception2:
    print('zero division')
else:
    print('all is ok')
finally:
    print('and this will run always')      # it has the same
effect after the block try :)
```

### 2.3.11

Fill in correct blocks:

```
_____:
    # commands
_____ Exception1:
    print('there must be integers on the entry')
_____ Exception2:
    print('division by zero')
_____:
    print('everything is ok')
_____:
    print('and this will be executed under all circumstances')
```

## 2.4 Exceptions generating

### 2.4.1

#### Creating an exception

Exceptions allow management and processing of unexpected situations that may occur during program execution.

However, it is not always sufficient to just handle exceptions that appear "by themselves", but in some cases it is necessary to create and generate exceptions,

most often within functions, to inform the main program that a problem has occurred.

Functions dealing with different tasks can be designed to respond to and deal with different exceptions in different ways.

We also know such approach in existing functions:

```
# by returning -1 value
ret = 'attention bad dog'
index = ret.find('where')
print(index)
```

**Program output:**

```
-1
```

```
# generating an exception
ret = 'attention bad dog'
index = ret.index('where')
print(index)
```

**Program output:**

```
ValueError
substring not found
```

## 2.4.2

Bypass generating an error and terminating the program so that the program does not break but returns -1

```
# generating an exception
ret = 'attention bad dog'
index = ret.index('where')
print(index)
```

It would probably be best to move the "risky" code to a separate function:

```
def newIndex(celyString, _____):
    _____:
        index = wholeString.index(looked_for)
        return _____
    _____:
        return _____

print(newIndex('attention bad dog', 'where'))
```

### 2.4.3

#### Raise

We use the **raise** keyword to create and raise an exception inside a function. It is used to explicitly announce that an exception has occurred, and then we can handle it within a function or let it pass to a higher level of the program. To raise an exception, we use the following syntax:

```
raise Exception("Exception description")
```

The exception type can be e.g. one of the known ones (ValueError, TypeError) and the exception description is any text we can display to the user.

The code in the function will react to the situation when division by zero occurs - we will not leave the generation of the exception to the system, but check the situation and create an exception with its own description:

```
def share(a, b):
    if b == 0:
        raise ZeroDivisionError("It is not possible to divide
by zero")
    return a / b
```

This exception can then be processed using the **try - except** construct.

```
try:
    result = share(10, 0)
except ZeroDivisionError as error:
    print(f"Error: {error}")
else:
    print(f"Result: {result}")
```

### 2.4.4

Fill in the code that generates **ValueError** in case of unsuccessful conversion.

```
def conversionDoInt(stringg):
    if not(stringg.isdigit()):
        _____ ("non-numeric input")
    else:
        return int(stringg)

try:
    a = conversionDoInt('aha')
    print(a)
```

```

    ValueError:
print('conversion error: ',error)

```

- raise
- try
- error
- ValueError
- as
- TypeError
- ValueError
- catch
- except
- to
- Exception

### 2.4.5

**Write a function that truncates a fraction. The input is the numerator and denominator values. If denominator is 0, throw an exception.**

In this case, returning a value of -1 or 0 as the result of fraction truncation would be inappropriate.

The best way to do this is to generate an exception and, if it is not treated in the body of the program, interrupt the execution of the program - this is logical, because wrong values might be used.

Again, we use the **raise** command in combination with some appropriate existing error type to create the exception:

```

def truncate(numerator, denominator):
    if denominator == 0:
        raise ValueError('Denominator is zero')
    a, b = numerator, denominator
    # finding the greatest common denominator
    while a != b:
        if a < b:
            b = b - a
        else:
            a = a - b
    numerator = numerator // a
    denominator = denominator // a
    return str(numerator) + ' / ' + str(denominator)

print(truncate(10,0))
print('end')

```

**Program output:**

```
ValueError
Denominator is zero
```

The program ends in the event of an error during this entry. We can therefore use the **try - except** block to catch and process the exception

```
try:
    print(truncate(10,0))
except Exception as err:
    print('An error occurred:',err)
print('Program continues...')
```

**Program output:**

```
An error occurred: Denominator is zero
Program continues...
```

 2.4.6

Arrange the lines of the program so that an error is caught if a non-existent string character is accessed when the following is called from the main program:

```
searchFrom("mum must prepare mutual food","m", 10, 20)
for i in range(start, end+1):
    print("the specified range has gone beyond the end of the string")
    if mychar == mystr[i]:
def searchFrom(mystr, mychar, start, end):
    try:
    except:
        print(f'Char {mychar} is at {i}')
```

 2.4.7**Multiple uses of raise**

There are situations where it is suitable to generate errors with different text for different incorrect values. An example can be the reaction of the `conversionDoInt()` method, which in the following form can react separately to entering an empty value and separately to all other errors.

```
def conversionToInt(mystring):
    if mystring == '':
        raise ValueError("empty string")
    if not(mystring.isdigit()):
        raise ValueError("non-numeric input")
```

```

else:
    return int(mystring)

try:
    a = conversionToInt('a')
    print(a)
except ValueError as error:
    print('conversion error: ',error)

```

**Program output:**

```
conversion error:  non-numeric input
```

In the previous case, we catch the error outside the function.

Alternatively, we have the possibility to capture and process an error directly in a function - if raise is between **try** and **except**, the generated exception will be captured by the nearest **except**.

```

def conversionDoInt(mystring):
    try:
        if mystring == '':
            raise ValueError("empty string")
        if not(mystring.isdigit()):
            raise ValueError("non-numeric input")
        else:
            return int(mystring)
    except Exception as error:
        return "captured in function: " + str(error)

a = conversionDoInt('a')
print("result:", a)

```

**Program output:**

```
result: captured in function: non-numeric input
```

## 2.4.8

Which statements are true for the following code?

```

def process(a, b):
    x = int(a)
    y = int(b)
    try:
        if y == 0:
            raise ValueError("division by zero")
        z = x / y
        print("the share is ", z)

```

```

except Exception as error:
    return error
else:
    return "ok"

try:
    a = input()
    b = input()
    x = process(a, b)
    print(x)
except:
    print("error")

```

- the value "ok" is returned if the text "share is " + value z is printed
- it applies that either the text "ok" is printed or the text "share is " + value z is printed
- if a conversion error occurs, it is caught by an except in the body of the main program
- if a conversion error occurs, it is caught by an except in the function body
- the part in the else branch in the function is never executed
- if b is '0' the text 'error' is printed
- the text "error" is printed if one of the input values is an empty string

## 2.5 Exceptions (programs)

### 2.5.1 Numeric values

Write a program that detects whether an integer value was entered correctly by catching an exception. In case of a correct value, the text "OK" is displayed, in case of an incorrect value, the text "Exception" is displayed.

**Input:** -268

**Output:** OK

**Input:** i am 5

**Output:** Exception

### 2.5.2 Age control

Write a program that includes a function that checks the name and age of a future employee, which will be loaded in the body of the main program. In the body of the function, if a name shorter than two characters is entered, generate a ValueError exception with the text "unspecified name", in case of an age under 18, a ValueError with the text "too young", and in case of an age over 70, a ValueError with the text

"too old". If all data is correct, write "accepted" in the body of the main program. If there is a non-numeric age on the input, catch the error and ensure the output of the text "problem with input".

E.g.

```
input: Johnny 25
output: accepted
```

```
input: x 20
output: unspecified name
```

```
input: Maria 12
output: too young
```

### 📖 2.5.3 Grade average

Write a program that calculates the average of a student's grades that are entered on a single line separated by spaces while the following requirements are met:

- Loading input from the user as a string separated by spaces.
- Splitting the string into individual grades based on spaces, in case of a problem, aborting the task and generating a `ValueError` with the text "non-numeric grade".
- Checking whether each grade falls within the range from 1 to 5 - in case of a problem, generating an "invalid grade" error.
- In case of an error, let its text be printed in the body of the program.
- In the case of an error-free calculation, the following is printed: average - value rounded to two decimal places.

E.g.

```
input: 5 4 3 2 1
output: 3.00
```

```
input: 4 3 5 2 1 6
output: invalid grade
```

```
input: 5 4 3 2 abc
output: non-numeric grade
```



# Files

## Chapter 3

## 3.1 Files

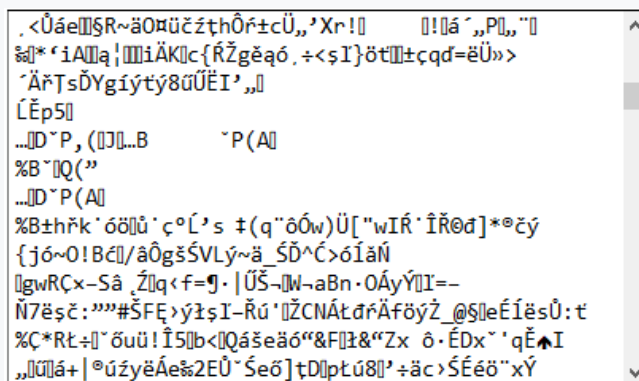
### 3.1.1

#### Files

Working with files is an important part of programming and provides programs with the ability to store and read data from various sources. Files in general can contain text, images, sounds or other data in a structured (organized) or general form.

**Binary files** are the first form of data storage and processing in computers. In binary files, data is stored in its natural form, which means that the written characters represent directly encoded numbers or text in the form in which they are stored in the computer's memory. This form is ideal for storing data that is not intended for human reading, but for fast and efficient computer processing.

They have e.g. the following form:



```
.<Úáe[]$R~äO#üčžtĥÔr±cÜ,,’Xr![]  []![]á´,,P[],”[]
%[]*´iA[]a[];[]i[]ĀK[]c{ŘŽgëaqó.÷<§I}ôtt[]±çqd=ëÜ»>
^ĀřTjSÖYgíýty8üŮĚI’,,[]
ĹĚp5[]
...D~P,([]]...B      ~P(A[]
%B~[]Q(”
...D~P(A[]
%B±hřk´ôö[]û´c°Ĺ’s ±(q“ôÓw)Ü[“wIR´ĪŘ@đ]*°čý
{jó~O!BĚ[]/âôgšŚVLý~ä_ŚĎ^Ć>óĪāŇ
[]gwRÇx-Sâ_Ž[]q<f=9·|ŮŠ-[]w-aBn·OÁyŸ[]I=-
Ň7ěšč:””#ŠFĚ>ýž§I-Rú’[]ŽCNÁłđřĀföýŽ_@š[]eÉĪěsŮ:t
%Ç*RŁ±[]”ôuü!Ī5[]b<[]Qāšeäó”&F[]ž&“Zx ô·ÉDx~’qĚ▲I
,,[]Ů[]á+|°úzyěĀe#2EŮ~Śeō]tD[]ptú8[]’÷äc>ŚÉéö”xÝ
```

#### Advantages:

- fast processing, the computer works with data in the form in which it is stored in its memory

#### Disadvantages:

- loading must always go into the same data structure
- if the file gets corrupted or a character is accidentally overwritten, the file is unusable.

### 3.1.2

Match pairs where the file extension matches the content type, for example "jpg" and "image".

- raster image (lossy compression) - \_\_\_\_\_

- raster image (lossless compression) - \_\_\_\_\_
- sound - \_\_\_\_\_
- compressed sound - \_\_\_\_\_
- a table with data separated most often by commas or semicolons - \_\_\_\_\_
- text file - \_\_\_\_\_
- document, probably MS Word - \_\_\_\_\_
  - xlsx
  - html
  - png
  - vob
  - docx
  - mp4
  - svg
  - csv
  - txt
  - css
  - wav
  - mp3
  - jpg

### 3.1.3

#### Text files

Text files represent a form of data organization that is closer to people. Currently, they are one of the most common ways of storing data. They are files that contain text in the form of character strings and can be read and edited by both humans and computers.

With today's fast hardware, it is usually possible to ignore the slowdown that occurs when text data is transformed from files to computer memory. More important than speed is readability and the ability to edit this data directly in the files.

It applies that:

- the data is stored in a "human readable" form
- they are often structured through special tags (XML, HTML, etc.)
- an overwritten or deleted random character usually has very little impact and the damage can be easily repaired
- work with text files is usually not done character by character, but line by line

There is no universal example for a text file, a typical one can be e.g. file with html, css and javascript commands determining the appearance of the document:

### 3.1.4

Which file types fall under the text file group?

- csv
- txt
- html
- css
- svg
- png
- bmp
- jpg
- pdf
- rtf

### 3.1.5

#### Text file

A text file represents a sequence of characters and special characters, while the most frequently used special character is probably the end of the line - `\n`.

The following file based on a user-friendly form:

```
monday
tuesday
wednesday
thursday
friday
saturday
sunday
```

... is actually stored in a file in the following form:

```
monday\ntuesday\nwednesday\nthursday\nfriday\nsaturday\nsunday
```

### 3.1.6

In how many lines are the data stored in the following file?

```
january\nfebruary\nmarch\na\npril
```

 3.1.7**Basic file operations**

Working with files usually means working with data stored in them.

In order to be able to read or write data, we first need to prepare the file for these activities:

- create if it does not exist and we want to insert data into it. Creating a file means creating a new file on disk. After a file is created, it is given a name and a location on disk, and can be used to store data.
- open it, if it already contains data and we want to read it, or add new ones. Opening a file establishes a connection between the program and the file on disk.
- after finishing the activities, we always need to close the file in order to release system resources and ensure the correct termination of operations with the file.

The operations we usually perform in a prepared (opened or created) file are:

- data entry
- reading data

 3.1.8

Complete the correct steps to calculate the grade point average of students stored in an existing file.

1. \_\_\_\_\_

2. \_\_\_\_\_

3. \_\_\_\_\_

- writing into the file
- file creation
- closing the file
- deleting the file
- reading from the file
- file creation

## 3.2 Reading and writing

### 3.2.1

#### Creating file

There is no special command for creating a file in Python. We always only open the file as such - for reading or writing. If we open the file for writing and it does not exist yet, it will be created.

Opening a file for notation can take the form of a programming command that contains the file name and mode specification ("w" for writing):

```
f = open('data.txt', 'w')
print(type(f))
```

#### Program output:

- the **f** variable provides a connection to the file - we will use it to call the appropriate commands
- the **open** command creates a connection between the variable representing the file (**f**) and the data in the file
- the connection will be directed to the '**data.txt**' file located in the same folder as the program
- the second parameter of the **open** command determines whether we open the file for writing (**w**) or reading (**r**)

### 3.2.2

Complete the parameters for opening the file:

Opening for writing:

```
f = open('data.txt', '_____')
print(type(f))
```

Opening for reading:

```
f = open('data.txt', '_____')
print(type(f))
```

### 3.2.3

#### Writing into the file

After opening the file for writing:

```
f = open('data.txt', 'w')
print(type(f))
```

... we can write data to the file, while there are two basic approaches:

- using the **print** command, where we enter a variable representing the open file as the **file** parameter

```
print('hello', file = f)
```

- using the **write** command, which is part of the file variable
- in this case, we also need to enter `\n` at the end of the line for delineation (in the case of **print**, it is inserted automatically)

```
f.write('hello again\n')
f.write('end')
```

- after finishing working with the file, we **must** close it

```
f.close()
```

- without closing the file, its contents will not be accessible to another program
- if we exit the program without closing, it may happen that the written characters **are not actually saved** in the file

### 3.2.4

Complete the program that stores the months of the winter semester (9-12) in separate lines. Make sure the last line is not empty.

```
f = open("data.txt", "w")
print("_____", file = f)
f.write("_____")
print("_____", file = f)
f.write("_____")
f._____( )
```

- november
- october
- close
- december
- october\n
- november\n
- september\n
- september
- december\n

### 3.2.5

#### Loading data from a file

We store the data in files so that we can read and process them later. Opening a file for reading is similar to opening a file for writing. By opening a file, we create a connection between our program and the target file.

When we open a file for reading, it allows us to read data from that file.

For our needs, we first create a file on the virtual computer by running the first code, and then we can work with it.

```
# writing
f = open('data.txt', 'w')
print('hello', file = f)
f.write('hello again\n')
f.write('end')
f.close()
# opening
f = open('data.txt', 'r')
line1 = f.readline()
print(line1)
line2 = f.readline()
print(line2)
line3 = f.readline()
print(line3)
f.close()
```

#### Program output:

```
hello
```

```
hello again
```

```
end
```

We usually read data from a file line by line using the **readline()** command. A line in a file is a list of all characters, including the line terminating character (**\n**), so lines are omitted from the printout.

In order not to skip a line unnecessarily, we will not delineate the **print** command - we will set the **end** parameter.

```
# opening
f = open('data.txt', 'r')
line1 = f.readline()
print(line1, end='')
```



```

line2 = f.readline()
print(line2, end='')
line3 = f.readline()
print(line3, end='')
f.close()

```

**Program output:**

```

hello
hello again
end

```

 3.2.6

Complete the program that reads three lines from the file data.txt and prints them next to each other separated by commas.

```

f = _____('data.txt', '_____')
line1 = _____()
print(line1, end=_____)
line2 = _____()
print(line2, end=_____)
line3 = _____()
print(line3, end=_____)
f._____()

```

- get
- close
- ""
- ""
- w
- f
- f
- f
- ''
- ;
- r
- get
- ""
- read
- ""
- read
- readline
- ''
- readline
- get
- read
- open
- readline

### 3.2.7

Usually, when we open a file, we have no idea how many lines it contains, and our goal is to process the entire file, i.e. loading data until we reach the end.

By repeatedly executing the **readline()** commands, we load each line, but if we don't know their exact number, we either cause an error or don't read the entire file.

It applies that the moment the **readline()** command returns an empty string, we have reached the end of the file.

We can use this fact when loading in a loop:

```
# preparing data
f = open('data.txt', 'w')
print('hello', file = f)
f.write('hello again\n')
f.write('end')
f.close()

# loading
f = open('data.txt', 'r')
line = f.readline()
while line != '':
    print(line, end='')
    line = f.readline()
f.close()
```

#### Program output:

```
hello
hello again
end
```

The condition of the **while** command will ensure that in the case of an empty file, the body of the loop does not run, and in the case of loading, the loop ends when an empty string is read.

### 3.2.8

Order the commands to load and printout the entire file correctly:

- while line != "":
- f.close()
- f = open('data.txt','r')
- line = f.readline()
- line = f.readline()

- `print(line, end='')`

### 3.2.9

In Python, if we can use the **while** loop for something, there is certainly a much more efficient solution for it through the **for** loop.

It is not otherwise in this case either. In the same way that we can navigate through the characters of a word or through a list of words split using **split()**, we can also navigate through a file. For the **for** loop, the elementary unit is a line.

So if we let the **for** loop go through the file, the loop variable represents the content of one line

```
# preparing data
f = open('data.txt', 'w')
print('hello', file = f)
f.write('hello again\n')
f.write('end')
f.close()

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

**Program output:**

```
hello
hello again
end
```

### 3.2.10

Complete the code to list the contents of the entire file:

```
_____ = _____ ('data.txt', 'r')
for _____ in _____:
    print(line, _____='')
f.close()
```

## 3.3 More operations

### 3.3.1

#### Exception treatment

Working with exceptions preceded working with files because files are a source of endless problems and potential errors.

So far we have ignored these risks, but they can easily backfire on us. What are the most common errors that can occur?

When **writing data**, we can:

- enter wrong name (file path)
- the file may become unavailable
- may run out of disk space

When **opening** a file:

- again, the path to the file may be bad
- we do not have reading permission
- access to the file will be lost, etc.

We will ensure the treatment with a try-except block, we will also print out the error:

```
# opening the file for reading will not allow writing
try:
    f = open('data.txt', 'r')
    print('hello', file = f)
    f.close()
except Exception as err:
    print('error: ', err)
```

**Program output:**

```
error: not writable
```

```
# attempt to open a non-existent file
try:
    f = open('data22.txt', 'r')
    for line in f:
        print(line, end='')
    f.close()
except Exception as err:
    print('error: ', err)
```

**Program output:**

```
error: [Errno 2] No such file or directory: 'data22.txt'
```

 3.3.2

Complete the code to catch exceptions that occur when loading the file:

```
_____:
f = open('data.txt', '_____' )
for line in _____:
    print(line, end='')
f._____( )
_____ Exception _____ err:
print('error:', err)
```

 3.3.3**Simplification of handling**

Simplifying file handling is one important time and error saver. In general, when working with files, it is recommended to use the with construct, which allows the program to open the file, work with it, and automatically close it when finished. Using the with construct eliminates the need to manually close files and ensures that the file is properly closed regardless of whether the operation succeeded or failed.

Of course, in order to prevent the program from being interrupted, it is necessary to use the **try-except** construct:

```
try:
    with open('data33.txt', 'r') as f:
        for line in f:
            print(line, end='')
    ## the file is automatically closed at this point
except Exception as err:
    print('error:', err)
print('Program continues despite the error')
```

**Program output:**

```
error: [Errno 2] No such file or directory: 'data33.txt'
Program continues despite the error
```

 3.3.4

Complete the code to load and print all lines from the file:

```

_____ open('data33.txt','r') _____ f:
_____ line _____ f:
_____ print(line, end='')

```

### 3.3.5

#### Loading the entire file

In some situations, we don't need to load the data line by line, but we need the whole text. We can get it as a string, while the ends of the lines are coded in the standard way (\n).

The code for loading the entire file is as follows:

```

# preparing data
f = open('data.txt','w')
print('hello', file = f)
f.write('hello again\n')
f.write('end')
f.close()

f = open('data.txt','r')
text = f.read()
print(text)
f.close()
print('length:', len(text))

```

**Program output:**

```

hello
hello again
end
length: 21

```

### 3.3.6

Match the commands to the appropriate situations:

- file opening - \_\_\_\_\_
- ending working with the file - \_\_\_\_\_
- loading the line - \_\_\_\_\_
- loading the whole file - \_\_\_\_\_
- writing into the file - \_\_\_\_\_
- line traversal - \_\_\_\_\_

- open
- writeln
- for

- get
- readline
- fsave
- read
- close
- if
- write
- fopen

### 3.3.7

#### Hidden characters

If we take the contents of the file, then the list of the number of characters is not quite accurate:

```
# entry - preparation
f = open('data.txt', 'w')
f.write('10\n')
f.write('20\n')
f.write('30')
f.close()

f = open('data.txt', 'r')
text = f.read()
print(text)
f.close()
print('length:', len(text))
```

**Program output:**

```
10
20
30
length: 8
```

If we want to also see hidden characters (e.g. `\n`), we use the `repr()` function:

```
f = open('data.txt', 'r')
text = f.read()
print(repr(text))
f.close()
print('length:', len(text))
```

**Program output:**

```
'10\n20\n30'
length: 8
```

`\n` represents one character - at the end of the third line there is no more.

 3.3.8

## Adding data to the end of the file

Adding data to the end of a file is a fairly common request. This operation is performed when we want to extend the contents of an existing file without overwriting the existing data. The ideal way to do this is to open the file for writing and set to the end of the content so we can append new data. In Python, we can use the 'a' parameter when opening a file, which will perform this operation automatically.

```
# file preparation
with open('data.txt','w') as f:
    print('start', file=f)
    for i in range(5):
        print(i, file=f)
    print('end', file=f)

# append
with open('data.txt','a') as f:
    print('post end', file=f)
    for i in range(2):
        print(i, file=f)

# control
with open('data.txt','r') as f:
    for i in f:
        print(i, end='')
```

**Program output:**

```
start
0
1
2
3
4
end
post end
0
1
```

As we can see, the added rows are actually placed after the data that was in the file originally.



 3.3.9

Complete the program so that, after opening, it detects how many lines the input file has and writes the detected number at the end of it.

```
with open('data.txt', 'r') as f:
    content = f.______()

print(______(content)) # let's write the text with hidden
characters
amount = content.count(_____) + _____ # assume there is no \n
after the last line

with open('data.txt', _____) as f:
    f.write(str(n))
```

## 3.4 Files (examples and programs)

 3.4.1

**Write a program that determines the number of words in a file.**

The task is quite simple, but it requires a sensible approach. It is apparent that:

- words are separated by spaces
- the problem is that we don't know if there is a space at the end of the line as well
- we need to read line by line and strip each line of spaces at the beginning and end of the line
- subsequently, through split, we find out the number of elements in the created list

```
# let's prepare the content file first
try:
    with open('long_text.txt','w') as f:
        print('Testing line 1 2 3 ',file=f)
        print('Python is a multi-paradigm language much like Perl,
unlike Smalltalk or Haskell. This means that instead of
forcing the programmer to use a certain programming style, it
allows the use of several. Python supports object-oriented,
structured, and functional programming. It is a dynamically
typed language, supports a large number of high-level data
types, and uses garbage collection for memory
management.',file=f)
```

```

    print('Although Python is often referred to as a
"scripting language", it is used to develop many large
software projects such as the Zope application server and the
Mnet and BitTorrent file sharing systems. It is also widely
used by Google. Proponents of Python prefer to call it a high-
level dynamic programming language, because the term
"scripting language" is associated with languages that are
only used for simple shell scripts or languages like
JavaScript: simpler and for most purposes less capable than
"real" programming languages like Python .',file=f)
    print('Another important feature of Python is that it is
easily extensible. New built-in modules can be easily written
in C or C++. Python can also be used as an extension language
for existing modules and applications that need a programmable
interface.',file=f)
except Exception as err:
    print('error: ',err)

```

... and solution:

```

number = 0
try:
    with open('long_text.txt','r') as f:
        for line in f: # reading line by line
            clearing = line.strip() # clearing the line
            number = number + len(clearing.split(' ')) # finding the
number of words separated by a space
except Exception as err:
    print('error: ',err)
print(number)

```

### 3.4.2

**Copy the content from one file to the other, with each line in the second file starting with the number of characters of the given line in the original file.**

This task is typical for presenting the principles of working with files in specific languages.

We can copy files with a single command, but usually there is also data processing involved

```

# text preparation
try:
    with open('demo.txt','w') as f:

```

```
    print('Testing line 1 2 3 ',file=f)
    print('Python is language.',file=f)
    print('It is widely used by Google.',file=f)
    print('It can be easily expanded.',file=f)
except Exception as err:
    print('error:', err)

# copying the entire file
try:
    with open('demo.txt','r') as fr, open('copy1.txt','w') as
fw:
        fw.write(fr.read())
except Exception as err:
    print('error:', err)

# solving the task
try:
    with open('demo.txt','r') as fr, open('copy2.txt','w') as
fw:
        for line in fr:
            new_line = str(len(line)) + ' - ' + line
            fw.write(new_line)
except Exception as err:
    print('error:', err)

# correctness check
try:
    with open('copy2.txt','r') as f:
        print(f.read())
except Exception as err:
    print('error:', err)
print('end')
```

**Program output:**

```
20 - Testing line 1 2 3
20 - Python is language.
29 - It is widely used by Google.
27 - It can be easily expanded.
```

end

### 3.4.3 First and last pupil

Write the code that reads the names of the pupils from the specified text file and prints the names of the first and last pupil of the list. Load the text file at the input.

```
Input : list.txt
Output:
Peter R.
Mira M.
```

```
Input : list2.txt
Output:
Miro V.
Brona A.
```

Preview of text file list.txt:

```
Peter R.
Miro V.
George L.
Beata G.
Andrea I.
Tom T.
Alena A.
Brona A.
Mira M.
```

**file1.py**

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here
    }
}
```

### 3.4.4 Names on B

Write the code that reads the names of the pupils from the specified text file and prints the names beginning with the letter B. Read the text file at the input.

```
Input : zoznam.txt
Output:
Beata G.
Brona A.
```

```
Input : zoznam2.txt
Output:
Bibiana A.
Bohus A.
```

Preview of text file zoznam.txt:

```
Peter R.
Miro V.
Juro L.
Beata G.
Andrea I.
Tester T.
Alena A.
Brona A.
Mira M.
```

file1.py

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here
    }
}
```

### 3.4.5 The best students

Write the code that lists students with an average grade of less than 1.5. At the input, read two text files that contain the students' names and their average grades. The average is separated by a semicolon in the file, in some numbers, a dot is used as a decimal separator, in some a comma and some are written as an integer. Print the names of all honoured students on the console (average  $\leq 1.5$ ). Print only names, not averages.

```
Input :
3A.txt
3B.txt
Output:
Peter R.
Miro V.
Andrea I.
Mira M.
Lolo L.
Miso K.
```

```

Input :
3A2.txt
3B2.txt
Output:
Miro V.
Andrea I.

```

Preview of text file 3A.txt:

```

Peter R. ;1.2
Miro V. ;1.3
Juro L. ;3,3
Andrea I. ;1,2
Tester T. ;3.0
Alena A. ;2.2
Mira M. ;1,5

```

Preview text file 3B.txt:

```

Lolo L. ;1.2
Miso K. ;1,3
Juro J. ;3.3

```

**file1.py**

```

public class JavaApp {

    public static void main(String[] args) {
        // write your code here
    }
}

```

### 3.4.6 Number of characters, lines, sentences and words

Write the code that detects how many characters, rows, sentences and words are contained in the specified text file. The name of the text file is given at the input. Suppose words do not divide at the end of a line, and no sentence ends with three dots. Print the following information to the console: "characters: 67 rows: 2 sentences: 9 words: 14".

```

Input : book.txt
Output: characters: 70 rows: 3 sentences: 5 words: 16

```

```

Input : book2.txt
Output: characters: 34 rows: 2 sentences: 2 words: 7

```

Preview of text file book.txt:

```
hi how are you I'm fine. And you?
This test is a test.
It tests itself!
```

**file1.py**

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here

    }
}
```

### 3.4.7 Maximum absence

Write the code that will find out in the given text file the name of the student with the most absence. At the input, is entered the file name that contains the student name in each line and a colon-separated number of absence hours. Read the data into an array that has 30 elements in size for up to 30 pupils. Print only the name of the student with the most absence on the console.

**Input :** data1.txt

**Output:** Anna

**Input :** data3.txt

**Output:** Lavonda

Preview of text file data1.txt:

```
Anna:55
Galen:10
Gustavo:20
Bethann:25
Rochel:0
Larhonda:15
```

**file1.py**

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here

    }
}
```

### 3.4.8 Calculation of absence

Write the code that finds the average number of absences in the specified text file. At the input, is given the file name that contains the student name in each row and a colon-separated number of absence hours. Print the number of registered pupils, the total and the average number of absences on the console. Round the number to one decimal place.

```
Input : data1.txt
Output:
10
122
12.2
```

Preview of text file data1.txt:

```
Anna:12
Jano:10
Peter:20
Adam:30
Mato:5
Jozo:15
Fero:16
Miro:4
Jana:7
Dana:3
```

```
file1.py
public class JavaApp {

    public static void main(String[] args) {
        // write your code here
    }
}
```

### 3.4.9 First and last alphabetically

Write the code that searches in the specified file and prints the names of the first and last pupils in alphabetical order. At the input, is given the name of the file containing the list of students (one name is given in each row). Use an array of max. size 10. Print the name of the first and last pupil in alphabetical order on the console.

```
Input :
list1.txt
Output:
```



```
Adam
Zuzana
```

Preview of text file list1.txt:

```
Jano
Peter
Anna
Adam
Mato
Jozo
Zuzana
Miro
Jana
Dana
```

**file1.py**

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here
    }
}
```

### 3.4.10 The longest name

Type the code that looks for the longest name in the specified file. At the input, is given the name of the file containing the list of students (one name is given in each row). Use an array of max. size 10. Print the longest name found on the console.

```
Input : list1.txt
Output: Kvetoslava
```

Preview of text file list1.txt:

```
Jano
Peter
Anna
Adam
Mato
Kvetoslava
Zuzana
Miro
Jana
Dana
```

**file1.py**

```
public class JavaApp {  
  
    public static void main(String[] args) {  
        // write your code here  
    }  
}
```

# Lists

## Chapter **4**

## 4.1 List

### 4.1.1

#### Lists

Lists are all around us, whether in everyday life or in programs. More than 90% of applications do not work with simple data, but with lists.

Typical lists you have already encountered include people, invoices, web addresses, measured values, etc.

The simplest list is a string of characters.

### 4.1.2

Select notations where we use variables as lists:

- `a = 42.8`
- `a = split(input())`
- `a = 'Warning it bites'`
- `a = int(input())`

### 4.1.3

With lists, we perform standard operations slightly different from working with simple variables:

- adding and deleting data
- browsing
- various calculations
- arrangement and so on.

The simplest operation is creating a list. Sometimes we create it by enumerating the elements.

The list represents a variable into which we insert a group of elements delimited by compound (square) brackets, while the elements are separated by commas:

```
temperatures = [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
students = ['Joseph Poppy', 'Alan Turing', 'Michal Pear',
            'Ivan Adam Barn']
years = [1945, 1969, 1971, 1978, 1980, 1984, 2012, 2015, 2019]
```

#### 4.1.4

Fill in the characters in the correct place so that the creation and filling of the list is correct:

```
authors = _____ 'Ernest Hemingway' _____ 'Erich Maria
Remarque' _____ 'Lev Nikolajevic Tolstoj' _____ 'Honore de
Balzac' _____
```

- ,
- .
- )
- :
- ;
- [
- }
- {
- (
- ,
- ;
- ]
- ;
- .

#### 4.1.5

Python also allows you to store elements of different types in a list, for example:

```
data = [20, 'Joseph', 14.8, True, 5000]
```

#### 4.1.6

Which entries for creating a list are correct?

- data = [20, 'Joseph', 14.8, True, 5000]
- data = [20, 'Joseph', 14.8 True, 5000]
- students = ['Joseph', 'Alan', 'Michal', 'Ivan']
- scientists = ['Milan', 'Rastislav', 'Joseph, 'Ivan']
- numbers = [1945, 1969, 1971, 1978, 1980, 19.84, 2012, 2015, 2019]
- temperatures= [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]

#### 4.1.7

The basic operation is to create an empty list. We often start the program by loading data into it - from the input, from a file or other sources.

To create an empty list, we use the entry:

```
mylist = []
```

or completely equivalent:

```
mylist = list()
```

#### 4.1.8

Complete the commands to create the lists:

```
my_list = _____()
my_second_list = _____ // use two characters
```

#### 4.1.9

The **print** command provides us with a very simple printout of the list:

```
my_list = ['Adam', 'Eva', 'Apple']
print(my_list)
```

**Program output:**

```
['Adam', 'Eva', 'Apple']
```

The printout contains elements separated by commas and enclosed in braces representing a list.

If we want to verify the type of the variable representing the list, we use the standard **type**:

```
a = []
b = list()
print('a is', type(a))
print('b is', type(b))
```

**Program output:**

```
a is
b is
```

#### 4.1.10

Retyping a list to a string is identical to printing it to the screen. How many characters does the printout have, or what value does the **print** command print?

```
data = ['Joseph', 'Poppy']
x = str(data)
print(len(x))
```

## 4.2 List elements

### 4.2.1

Access to list elements is provided through indexes. The same rules apply as for a string:

- the first element has index 0
- the last element has an index one less than the number of elements

```
a = ['January', 'February', 'March', 'April', 'May', 'June',  
'July', 'August', 'September', 'October', 'November',  
'December']  
print(a[0])  
print(a[11])  
print(a[-2])
```

**Program output:**

```
January  
December  
November
```

### 4.2.2

Which days are **NOT** printed in the following list?

```
my_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',  
'Friday', 'Saturday', 'Sunday']  
print(my_list[1])  
print(my_list[2])  
print(my_list[4])  
print(my_list[-3])  
print(my_list[-5])
```

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday
- Sunday

### 4.2.3

The elements in the list can be changed by simply assigning a new value to a variable with a given index.

```
my_list[i] = value
```

Each element of the list behaves as a separate variable, it is even possible to change the type of the element at the specified position:

```
a = ['January', 'February', 'March', 'April', 'May', 'June']
a[0] = 'January'
a[1] = 'II.'
a[2] = 3.0
print(a)
```

**Program output:**

```
['January', 'II.', 3.0, 'April', 'May', 'June']
```

#### 4.2.4

How many numeric values will the list contain after the following code is executed?

```
a = ['January', 'February', 'March', 'April', 'May', 'June']
a[0] = 'January'
a[1] = len(a[1])
a[2] = str(a[1])
if (a[1] * a[2]) < 'a':
    a[4] = a[1]
```

#### 4.2.5

The already known function **len** returns the number of elements in the list:

```
months = ['January', 'February', 'March', 'April', 'May',
'June', 'July', 'August', 'September', 'October', 'November',
'December']
data = [20, 'Joseph', 14.8, True, 5000]
print(len(months))
print(len(data))
```

**Program output:**

```
12
5
```

It is useful e.g. if we are loading a character-separated list from the input and we need to know the number of its elements.

We have already solved such a request through the **split()** function.

Now we can say that the result of the **split()** function is a **list**:



```
data = input('Enter a comma-separated list').split(',')
print(data)
print('The number of elements in the list is', len(data))
```

**Program output:**

```
Enter a comma-separated list I,am,here,for,a,holiday['I',
'am', 'here', 'for', 'a', 'holiday']
The number of elements in the list is 6
```

 4.2.6

What is the output of the following program?

```
listt = ['Monday', 'Wednesday', 'Thursday', '', 'Saturday']
listt2 = [1, 2, 7, 6, 2.8, 3]
print(len(listt) - len(listt2))
```

 4.2.7**Traversing the list**

The operations we perform on a list usually require processing each element. We use a loop to access each element.

We implement the traversal through a cycle from the first element to the last one located at position `len(list) - 1`:

```
my_list = ['Adam', 'Berta', 'Cecil', 'Dana', 'Ema', 'Fero',
'Gusto', 'Hana']
for i in range(0, len(my_list)):
    print(i, my_list[i])
```

**Program output:**

```
0 Adam
1 Berta
2 Cecil
3 Dana
4 Ema
5 Fero
6 Gusto
7 Hana
```

A simpler alternative, if we are not interested in the sequence number of the element, is to traverse the elements of the list:

```
my_list = ['Adam', 'Berta', 'Cecil', 'Dana', 'Ema', 'Fero',
'Gusto', 'Hana']
for item in my_list:
```

```
print(item, 'with the length of', len(item))
```

**Program output:**

```
Adam with the length of 4
Berta with the length of 5
Cecil with the length of 5
Dana with the length of 4
Ema with the length of 3
Fero with the length of 4
Gusto with the length of 5
Hana with the length of 4
```

### 4.2.8

What will be stored in the `cnt` variable after the program ends?

```
my_list = ['Augustina', 'Branislava', 'Cecil', 'Dana',
'Emanuela', 'Frantisek', 'Gustav', 'Hana']
cnt = 1
for element in my_list:
    if len(element) > len(my_list):
        cnt += 1
print(cnt)
```

### 4.2.9

## Enumerate

A special function that modifies the list to ensure its indexing is `enumerate()`. It allows us to skip working with `range` and `len`, but its purpose is exactly the same.

Using `enumerate` creates an index and element pair from the original list. We write this pair after the `for` command as follows:

```
people = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
for index, person in enumerate(people):
    print(index, person)
```

**Program output:**

```
0 Peter
1 Pavol
2 Michael
3 Juraj
4 Jan
```

The variable that will represent the index is written after the `for` command as the first (**index**) and the variable that will represent the list element as the second (**person**).

We include **enumerate** in the range of the loop. List represents a variable of type **list**.

The advantage of **enumerate** is that it allows us to set the start of the numbering. That is, if we want the first element to appear in the loop as an element with index 1 and not 0, we can use the entry:

```
people = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
for index, person in enumerate(people, start = 1):
    print(index, person)
```

**Program output:**

```
1 Peter
2 Pavol
3 Michael
4 Juraj
5 Jan
```

#### 4.2.10

Complete the commands so that the program lists the serial number of the element, the element and the number of its characters:

```
students = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
for index, _____ in _____(students, _____ = 1):
    print(_____ (index) + '. ' + person + ' - ' +
          _____ (_____ (person)) + ' characters')
```

- int
- len
- numeral
- person
- begin
- start
- str
- len
- students
- str
- enumerate
- int

## 4.3 List editing

### 4.3.1

#### Loading list elements

By far the easiest way to populate a list by the user is by reading the character-separated elements in the input and then splitting them with **split()**:

```
data = input('Enter a comma-separated list').split(',')
print(data)
```

#### Program output:

```
Enter a comma-separated list 1,2,3,4,5,6['1', '2', '3', '4',
'5', '6']
```

However, for various reasons, we need to add new elements to such a list at any moment of the program's execution.

Adding a new element is ensured by using the **append()** command, which adds the element to the end of the list.

```
data.append('new')
print(data)
```

#### Program output:

```
['1', '2', '3', '4', '5', '6', 'new']
```

### 4.3.2

Arrange the variables so that the result of the program is a list with elements:

```
['one', 'lady', 'said', 2, 'things', 'for', '100', 'percent']
```

```
number1 = 100
number2 = 2
string1 = 'percent'
string2 = 'na'
listt = ['warning', '_____', 'dog']
listt = ['_____', 'lady', '_____']
listt.append(_____)
listt.append(_____)
listt.append(_____)
listt.append(_____)
listt.append(_____)
print(listt)
```

- 'lady'
- string2
- bad
- number2
- string1
- 'things'
- number1
- said
- one

### 4.3.3

Already in the case of working with strings, we used commands separated from the variable by a dot.

```
my_string = 'Mama has Ema'
new = my_string.upper()
print(new)
```

**Program output:**

```
MAMA HAS EMA
```

```
my_string = 'Mama has Ema'
new = my_string.split()
print(new)
```

**Program output:**

```
['Mama', 'has', 'Ema']
```

Similarly, the **list** structure is programmed in such a way that makes the functions that are part of it available to the programmer, and these functions manipulate the data stored in the given structure. Functions are separated from the specific variable representing the list by a dot and we refer to them as **methods**.

So an example for a **list** is:

```
my_list.append('new')
```

### 4.3.4

Which of the following notations contain methods?

- if my\_string.isdigit():
- my\_string.strip()
- listt.append('aha')
- x = len(listt)
- range(4,10,2)
- x = listt[3]
- x = list()

 4.3.5

## Alternative addition

We can also add an element to the end of the list by creating a one-element list from it using `[]` and adding it to the original list using the `'+'` operation.

```
my_list = [1, 2, 3, 4, 5]
value = 10
my_list = my_list + [value] + [8] # correct
print(my_list)
```

**Program output:**

```
[1, 2, 3, 4, 5, 10, 8]
```

Attention, the following notation is **incorrect**, you cannot combine the list type and the number type:

```
my_list = [1, 2, 3, 4, 5]
new = 10
my_list = my_list + new # incorrect
print(my_list)
```

**Program output:**

```
TypeError
can only concatenate list (not "int") to list
```

 4.3.6

Sort the rows to get the list `[1, 2, 3, 4, 5, 6, 7, 8]`

- `b = b + [8]`
- `a = [3, 4]`
- `b = a + [5, 6]`
- `a = b + [7]`
- `b = [1] + a`
- `b = [2] + b`
- `print(b)`

 4.3.7

## Sequential loading of list elements

Elements can be added to the list gradually from the input.

The solution is most often started by creating an empty list.

If we decide that one input will load one element from the list, we load using the `input()` command.

It is also necessary to define the situation when loading should end. Considering that we usually don't know the number of elements (we could enter them all at once in a line separated by a suitable character), we need to define a condition when the loading should stop.

We do so by means of a terminating element, e.g. '0', 'x', '-1' and so on. If we get this character from the input, we end the loading.

```
my_list = []
while True:
    element = input()
    if element == '0':
        break
    my_list.append(element)
print(my_list)
```

**Program output:**

```
10 20 0['10', '20']
```

### 4.3.8

Fill in the commands to load the list from the input. Let the 'x' sign ensure the end of loading:

```
my_list = _____
while _____:
    element = input()
    if element == '_____':
        _____
    my_list._____(element)
print(my_list)
```

- continue
- -1
- x
- True
- 0
- stop
- ()
- append
- list()
- break
- False
- return

 4.3.9

## Element location

If we want to place the element in a specific position, and not at the end of the list, we use the command:

```
insert(position, content)
```

The command inserts the given element at the indicated position and thus moves the elements located behind the given position:

```
my_list = [1, 2, 3, 4, 5]
my_list.insert(2, 'new')
print(my_list)
```

**Program output:**

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

 4.3.10

What will the list look like after executing the following commands?

```
my_list = [1, 2, 3, 4, 5]
my_list.insert(2, 'white')
my_list.insert(2, 'green')
my_list.insert(4, 'blue')
my_list.insert(1, 'red')
my_list.insert(3, 'yellow')
print(my_list)
```

Result:

```
[_____, 'red', _____, _____, _____, _____, _____, _____, _____]
'yellow'
3
1
4
5
'green'
2
'white'
'blue'
```



 4.3.11

## Deleting an element

We can use two approaches to delete an element from the list:

Using the **del()** command, which deletes the element at the specified position:

```
my_list = [1,2,3,4,5,6]
del(my_list[3])
print(my_list)
```

**Program output:**

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

Using the **remove()** method, which finds the first occurrence of an element with the specified content in the list and deletes it.

If the specified element does not exist in the list, it will generate an exception.

```
my_list = [1,2,0,3,4,0,5,0,6]
my_list.remove(0)
print(my_list)
my_list.remove(0)
print(my_list)
my_list.remove(0)
print(my_list)
my_list.remove(0)
print(my_list)
```

**Program output:**

```
[1, 2, 3, 4, 0, 5, 0, 6]
[1, 2, 3, 4, 5, 0, 6]
[1, 2, 3, 4, 5, 6]
```

**ValueError**

```
list.remove(x): x not in list
```

 4.3.12

Arrange the commands so that the resulting list looks like [1, 2, 3, 4, 5].

```
my_list = [1,2,0,3,4,0,5,0]
```

- del(my\_list[2])
- del(my\_list[5])
- my\_list.remove(0)

## 4.4 List (programs)

### 4.4.1 First and the last

For the specified list of names separated by semicolons, write the first and last name below, e.g.:

```
Input: Ivan;Jan;Michal;Jozef
Output:
first: Ivan
last: Jozef
```

### 4.4.2 Average grade

The input contains a list of grades separated by spaces. Check the list for invalid items or items outside the range 1-5 and calculate the average grade rounded to two decimal places. If there are incorrect values in the list, do not calculate the average, but print: incorrect input.

```
Input: 5 4 3 2 1
Output: 3.0
```

```
Input: 5 0 3 2 1
Output: incorrect input
```

```
Input: 5 accident 3 2 1
Output: incorrect input
```

### 4.4.3 Number of occurrences

At the input, a list of names separated by commas is entered in one line. The search name is entered in the second line. Write a program that prints all the positions in the list where the given name is found and at the end prints their total number.

```
Input:
Peter,John,Michael,John,Paul,John
John
Output:
2 4 6, number=3
```

```
Input:
Peter,John,Michael,John,Paul,John
Ivan
Output:
, number=3
```

#### 4.4.4 Clearing

The input contains a list of elements, among which are numerical values (even decimal) and non-numeric, usually text values. They are separated by a comma followed by a space. Write a program that creates a new list containing only numeric values and prints it.

```
input: 2, text, 3.5, 4, hello
output: [2, 3.5, 4]
```

```
input: one, two, three, four, five
output: []
```

#### 4.4.5 Dividing the list

Write a program that divides numerical data from the input separated by a comma and a space into two lists: positive, negative according to the sign of the corresponding number. Ignore the value 0. At the end of the program, write the lists elements below one another. For example

```
input: 2, -5, 0, 7, -3, 0, 10, -8
output:
positive: [2, 7, 10]
negative: [-5, -3, -8]
```

#### 4.4.6 Days and night

Write a program that, for the temperatures of the days during the month entered in the form of day/night as decimal numbers and separated by a pair of comma+space characters, finds and prints the following:

- average daily temperature rounded to two decimal places
- average night temperature rounded to two decimal places
- number of tropical days (days where the temperature is above 30 degrees)
- number of tropical nights (nights where the temperature is above 20 degrees)
- the number of arctic days (the temperature during the day and night does not rise above -10 degrees).

```
Input: 3.0/-1.0, 2.0/-3.0, 0.0/-5.0, 1.0/-2.0, 0.0/-3.0, -
1.0/-4.0, -2.0/-5.0, -3.0/-6.0, -4.0/-7.0, -5.0/-8.0, -6.0/-
9.0, -7.0/-10.0, -8.0/-11.0, -9.0/-12.0, -10.0/-13.0
Output:
ADT: -3.27
ANT: -6.6
```

```

NTD: 0
NTN: 0
NAD: 1

```

### 4.4.7 Hiking

At the input, a list of altitudes of places on the hiking trail, separated by commas + spaces, is entered. Find out how many going up and down there are between individual points - consider each pair of points separately. Calculate the total height of climbs and the total height of descents.

```

Input: 100, 150, 200, 180, 220
output:
Uphill: 3
Downhill: 1
Overall uphill: 140
Overall downhill: 20

```

### 4.4.8 Too long names

The input contains a list of names separated by spaces in one line. Create two lists: names, long names. In the first list, put all names with a number of characters less than or equal to the value given in the input in the second line. Place names longer than this value in the second list. As the output of the program, write both lists in the form:

names: [..., ...]

long names: [..., ...]

```

input:
Adam Eva John Dora Samantha Alexander
6
output:
names: ['Adam', 'Eva', 'John', 'Dora']
long_names: ['Samantha', 'Alexander']

```

### 4.4.9 Abbreviation of names

The input contains a list of names separated by spaces in one line. The second line shows the maximum number of accepted characters. Edit the list so that if the name is longer than the specified number of characters, the program cuts off the remaining characters and replace them with three dots.

Print the modified names separated by spaces as output.

```
names: [..., ...]
```

```
long names: [..., ...]
```

```
input:
Adam Eva John Dora Samantha Alexander
6
output:
Adam Eva John Dora Samant... Alexan...
```

#### 4.4.10 Progressive loading

Write a program that divides numerical data from the input, read sequentially from separate lines, into two lists: positive, negative according to the sign of the corresponding number. The value 0 at the input indicates the end of loading. At the end of the program, write the average of the positive and the average of the negative list below each other, rounded to one decimal place. If the list is empty, write 0.0 as its average. For example:

```
input:
5
-3
8
-2
0
output:
The average of positive numbers: 6.5
The average of negative numbers: -2.5
```

# List Processing

Chapter **5**

## 5.1 Creating a list

### 5.1.1

#### List loading - specifics

In many tasks, we need to convert the list obtained from the input to an integer or decimal number, which we then insert into a new list.

For data loaded from input and separated by commas, the code might look like this:

```
my_list = input().split(',') # e.g. '10','20','30','40'
numbers = []
for x in my_list:
    numbers.append(int(x))
print(numbers)
```

**Program output:**

```
1,2,3,8[1, 2, 3, 8]
```

- or

```
my_list = input().split(',') # e.g. '10','20','30','40'
numbers = []
for x in my_list:
    numbers += [int(x)]
print(numbers)
```

**Program output:**

```
1,2,3,8[1, 2, 3, 8]
```

With a universal solution, we also need to take into account the possibility that the input can be empty. The result of the **split()** method is always a list, so in the condition we check if it is not empty:

```
my_list = input().split(',') # e.g. '10','20','30','40'
numbers = []
if my_list != []:
    for x in my_list:
        numbers.append(int(x))
print(numbers)
```

**Program output:**

```
1,2,3,8[1, 2, 3, 8]
```

### 5.1.2

Complete the foolproof code so that it reads data from the input. Make sure that if it reads incorrect input, it simply skips it and continues:

```

my_list = input().split(',') # e.g. '10','20','30a','40'
numbers = []
if my_list != ____:
    for x in my_list :
        ____:
            numbers.append(____)
        ____:
            ____
print(numbers)

```

- try
- continue
- x
- []
- [""]
- int(x)
- break
- "
- except
- stop

### 5.1.3

## Merging lists

We already know that lists can be joined using '+'.

Merging is a concatenation of lists into a sequence of elements copying the order in which the lists were added

```

a = [1,1,1]
b = [2,2]
c = [3,3,3,3]
x = a + b
y = c + a + b
print(x)
print(y)

```

**Program output:**

```

[1, 1, 1, 2, 2]
[3, 3, 3, 3, 1, 1, 1, 2, 2]

```

Multiplying a list by a numeric constant works similarly to addition:

```

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

```



**Program output:**

```
[1, 2, 1, 2, 1, 2]
```

### 5.1.4

What will be the result of the following entry?

```
a = [5, 1]
b = [3, 1]
c = a + 2 * b + a
print(c)
```

### 5.1.5

## Creating a list

We already know that there are two ways to create an empty list:

```
a = []
b = list()
```

If we want to create a list with contents, we can use functions that result in a list.

For integers **range()** with suitable range and step, e.g.:

```
a = list(range(1,10,3))
print(a)
```

**Program output:**

```
[1, 4, 7]
```

For a list of characters transformation of a string using the **list()** function:

```
a = list('Aladin')
print(a)
```

**Program output:**

```
['A', 'l', 'a', 'd', 'i', 'n']
```

If the list is filled with elements of another list, we can create an independent copy of it:

```
a = ['mathematics', 'I.T.', 'physics', 'chemistry']
b = list(a)
print(b)
```

**Program output:**

```
['mathematics', 'I.T.', 'physics', 'chemistry']
```

 5.1.6

Which entries for creating a list are correct:

- `a = []`
- `a = list()`
- `a = list('Mama has Emma')`
- `a = split('warning')`
- `a = 'warning'.split()`
- `a = {10,20,30}`
- `a = [10, 20, 30]`

 5.1.7

### Non-iterable objects

Besides the previous entries, where the error was usually in the logic, there are other reasons why a variable or list cannot be used to create a sequence of elements. This is because some elements are not iterable:

```
my_list = list(10)
```

**Program output:**

```
TypeError
'int' object is not iterable
```

```
my_list = list(True)
```

**Program output:**

```
TypeError
'bool' object is not iterable
```

```
my_list = list(1,3,2)
```

**Program output:**

```
TypeError
list expected at most 1 argument, got 3
```

 5.1.8

Which entries for creating a list are correct:

- `a = list('a')`
- `a = list()`
- `a = list([10,20,30])`
- `a = list(10,20,30)`
- `a = list(4.2)`
- `a = list(4 <|3)`
- `a = list('try')`

### 5.1.9

#### Creating a list from a file

When working with a file, we use the **for** loop to iterate over its individual lines. We can obtain these lines and directly assign them to a list of type **list()** by simple assignment:

```
try:
    lines = list(open('data.txt', 'r'))
    print(lines)
except Exception as err:
    print('error:', err)
```

Program output:

```
['english\n', 'mathematics\n', 'chemistry\n', 'biology\n']
```

For the code to work, of course, we need to have the corresponding file created:

```
# data preparation
try:
    with open('data.txt','w') as f:
        print('english',file=f)
        print('mathematics',file=f)
        print('chemistry',file=f)
        print('biology',file=f)
except Exception as err:
    print('error:', err)
```

### 5.1.10

Complete the code to transform the lines of the file into a list:

```
_____:
    lines = _____(_____('data.txt', 'r'))
    print(lines )
except Exception _____ err:
    print('error:', err)
```

## 5.2 Processing elements

### 5.2.1

#### Operations above the list

Python provides several functions that work with a list of list elements. It handles these values in accordance with the data types that the lists contain. As long as all the elements are of the same type, we can use the sum, min and max operations, which have different behavior defined for the different data types of the elements in the list:

- **sum (list)** returns the sum of the elements stored in the list
- **max (list)** returns the element with the maximum value
- **min (list)** returns the element with the minimum value

```
my_list = [1,2,3,4,5,0,8]
print(sum(my_list))
print(min(my_list))
print(max(my_list))
```

**Program output:**

```
23
0
8
```

```
my_list = ['1','2','3','4','5','0','8']
# print(sum(my_list))
print(min(my_list))
print(max(my_list))
```

**Program output:**

```
0
8
```

```
my_list = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
# print(sum(my_list))
print(min(my_list))
print(max(my_list))
```

**Program output:**

```
Jan
Peter
```

The **sum** operation is not defined for text values.

In the case of a list containing a combination of numeric and text values, it is not possible to implement these operations, because Python does not support data comparison between different types.

```
my_list = [1, '2', '3', 4, '5', '0', '8']
# print(sum(my_list))
print(min(my_list))
print(max(my_list))
```

Program output:

```
TypeError
'<' not supported between instances of 'str' and 'int'
```

### 5.2.2

What does the following program print?

```
my_list = ['25', '13', '24', '5.0', '110', '8']
x = min(my_list)
print(x)
```

### 5.2.3

What does the following program print?

```
my_list = [25, 13, 24, 5, 110, 8]
x = sum(my_list)
print(x)
```

### 5.2.4

## Searching for a value in a list

To browse the list and e.g. searching for a specific element we can use looping through the list, but we also have more efficient tools at our disposal.

The easiest way to ask for the existence of an element in the list is to use in:

```
my_list = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
boyfriend = 'Juraj'

if boyfriend in my_list:
    print(boyfriend, 'is there')
else:
    print(boyfriend, 'is not there')

ex = 'Jozef'
if ex in my_list:
    print(ex, 'is there')
else:
```

```
print(ex, 'is not there')
```

**Program output:**

```
Juraj is there
Jozef is not there
```

### 5.2.5

What does the program print?

```
my_list = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
boyfriend1 = 'Juraj'
boyfriend2 = 'Pavo'
print(boyfriend1 in my_list)
print(boyfriend2 in my_list)
print(boyfriend2 not in my_list)
print('Jan' not in my_list)
```

- True, False, True, False
- error
- True, True, True, False
- True, True, True, True
- False, True, False, True

### 5.2.6

#### Occurrence in the list

If it is not enough for us to find out whether the item is in the list, but we also want to find out the position or the number of occurrences, we can use the following functions:

- **index** - finds the position of the first occurrence of the element in the list
- **count** - finds the number of occurrences of the elements in the list

```
my_list = ['Peter', 'Jan', 'Michael', 'Juraj', 'Jan']
boyfriend = 'Juraj'
print(my_list.index(boyfriend))
print(my_list.count('Jan'))
```

**Program output:**

```
3
2
```

### 5.2.7

What will be the result of the following code?

```
my_list = list('SAgarmatha')
x = my_list.count('a')
print(x)
```

### 5.2.8

If the searched element is not in the list, it generates an exception:

```
my_list = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
boyfriend = 'Daniel'
try:
    print(my_list.index(boyfriend))
except:
    print('not found')
```

**Program output:**

```
not found
```

The `index()` function also has two additional optional parameters:

```
i = my_list.index(search, start, end)
```

- where **start** is the position from which to start the search
- **end** is the position before which the search should end

Examples of use are presented in the following code:

```
my_list = [0,1,2,3,4,5,6,7]
value = 4

try:
    print(my_list.index(value))
except:
    print(value, 'not found')

try:
    print(my_list.index(value,2)) # starts searching from the
index 2
except:
    print(value, 'not found')

try:
    print(my_list.index(value,5)) # starts searching from the
index 5
except:
    print(value, 'not found from 5')
```

```
try:
    print(my_list.index(value,2,4)) # starts searching from index
    2 and ends before 4
except:
    print(value, 'not found 2,4')
```

**Program output:**

```
4
4
4 not found from 5
4 not found 2,4
```

### 5.2.9

What value must be put into the variable x to print the value 5?

```
my_list = [0,1,5,2,3,3,6,3,5]
x = ???
i = my_list.index(3, x+1, 6)
print(i)
```

### 5.2.10

## Arrangement of elements

A common and useful operation is the arrangement of elements. This will run over the list if it contains values of the same type.

There are two alternatives:

- the list sorter **sort()** method sorts the elements of the list it is applied to
- the **sorted()** function returns a new list as the result of its operation, leaving the original list unchanged

```
my_list = [3, 1, 5, 2, 4]
my_list.sort()
print(my_list)
```

**Program output:**

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

```
my_list = [3, 1, 5, 2, 4]
new = sorted(my_list)
print(my_list)
print(new)
```

**Program output:**

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



 5.2.11

Arrange the elements as they will be arranged in the list variable after the program ends:

```
my_list= ['anna', 'Beata', 'cynthia', 'Daniel', 'eva',
'Filip']
my_list.sort()
print(my_list)
```

- Daniel
- anna
- Beata
- eva
- cynthia
- Filip

 5.2.12

## Other list operations

- **pop()** - removes the last element from the list and returns it as the result of its operation

```
my_list = [1,2,0,3,4,0,5,0,6]
last = my_list.pop()
print('last:',last)
print(my_list)
```

**Program output:**

```
last: 6
[1, 2, 0, 3, 4, 0, 5, 0]
```

- **pop(index)** - removes the element at the specified position from the list and returns it as the result of the call

```
my_list = ['Anna', 'Hana', 'Johanna', 'Klementina',
'Viktoria']
second = my_list.pop(1)
print('second:',second)
print(my_list)
```

**Program output:**

```
second: Hana
['Anna', 'Johanna', 'Klementina', 'Viktoria']
```

- **clear()** - clears the entire list

```
my_list = [1,2,0,3,4,0,5,0,6]
second = my_list.clear()
print('second:',second)
print(my_list)
```

**Program output:**

```
second: None
[]
```

### 5.2.13

Arrange the elements as they will be stored in the list after the execution of the following sequence of commands:

```
my_list = ['Anna', 'Hana', 'Johanna', 'Klementina',
           'Viktoria']
x = my_list.pop(1)
y = my_list.pop()
my_list.append('Anna')
x = my_list.pop(3)
my_list.append(x)
my_list.append(y)
print(my_list)
```

- Anna
- Viktoria
- Johanna
- Anna
- Klementina

## 5.3 Slices

### 5.3.1

#### Slices

The functionality already known from the use of text strings allows you to get a part or subparts from the list based on their indexing.

The syntax for slices in Python basically consists of using square brackets [] where we define the range of values we want to retrieve from an existing list.

The range is specified using indexes that include a start and end index separated by a colon. Sometimes it can be supplemented with a third value expressing the step.

```
new = my_list[start : stop : step]
```

Examples:

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[1:4]
print(new)
```

**Program output:**

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

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[1:6:2]
print(new)
```

**Program output:**

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

### 5.3.2

Complete the parameters so that you get the names Peter, Karolína, Mária, Matej.

```
birth_names = ["Adam", "Eva", "Ján", "Zuzana", "Peter",
               "Karolína", "Mária", "Matej", "Laura", "Michal"]
names = birth_names[____:____]
print(names)
```

### 5.3.3

Complete the parameters so that you get the names Ján, Karolína, Laura.

```
birth_names = ["Adam", "Eva", "Ján", "Zuzana", "Peter",
               "Karolína", "Mária", "Matej", "Laura", "Michal", "Ivan",
               "Zdena"]
names = birth_names[____:____:____]
print(names)
```

### 5.3.4

We can also use negative indexes. Do not forget that when using them, they are always first converted to positive indices, and if we want to go from larger to smaller, it is necessary to set the step to a negative value.

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[-1:-3]
print(new)
```

**Program output:**

```
[ ]
```

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[-3:-1]
print(new)
```

**Program output:**

```
[5, 6]
```

Usually, the purpose of using negative indexes is also to change the order of elements from the list. In that case, we enter a negative step:

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[-1:-3:-1]
print(new)
```

**Program output:**

```
[7, 6]
```

### 5.3.5

Complete the parameters so that you get the names Laura, Matej, Mária. Use negative indexes.

```
birth_names = ["Adam", "Eva", "Ján", "Zuzana", "Peter",
               "Karolína", "Mária", "Matej", "Laura", "Michal", "Ivan",
               "Zdena"]
names = birth_names[_____:_____:_____]
print(names)
```

### 5.3.6

Complete the parameters so that you get the names Peter, Ján, Adam. Use negative indexes.

```
birth_names = ["Adam", "Eva", "Ján", "Zuzana", "Peter",
               "Karolína", "Mária", "Matej", "Laura", "Michal", "Ivan",
               "Zdena"]
names = birth_names[_____:_____:_____]
print(names)
```

### 5.3.7

As in the case of strings, we can also omit some slice parameters when working with lists. In that case, they are filled in automatically - the smallest possible value for the first position, the largest for the second.

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[:5:]
print(new)
```

**Program output:**

```
[0, 1, 2, 3, 4]
```

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[::-1]
print(new)
```

**Program output:**

```
[7, 6, 5, 4, 3, 2, 1, 0]
```

### 5.3.8

Which elements will the new list contain, defined as follows:

```
my_list = [0,1,2,3,4,5,6,7]
new = my_list[:3:]
print(new)
```

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7

### 5.3.9

#### Betrayal of the iterator

How are the following programs different?

```
my_list = [1,1,1,1,1,1]
for x in my_list:
    x += 1
print(my_list)
```

**Program output:**

```
[1, 1, 1, 1, 1, 1]
```

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

**Program output:**

```
[2, 2, 2, 2, 2, 2]
```

In both tasks, we change the value of the element in the list, with which we are currently working within the loop.

- In the first example, we only change the value of the variable that represents the element of the list - the control of the cycle inserts the value of the corresponding element in the list into it, but this variable is no longer linked to the elements of the list. It means that any change of its value **will not affect** the elements of the list.
- In the second case, we directly change the content of the i-th element of the list.

The conclusion that follows from this difference is that if we want to change the elements of the list in a cycle, we need to access the elements of the list directly (via the corresponding index).

 5.3.10

Complete the code so that all even values of the list are doubled:

```
my_list = [1, 2, 3, 3, 2, 4]
for index _____ in _____ (List):
    if x _____ 2 == 0:
        my_list[_____] = 2 * x
print(my_list)
```

## 5.4 List comprehension

 5.4.1

### List comprehension

List comprehension is a way to create a new list from an existing one using short entry. It requires specifying the rules for creating new list elements. This notation makes it possible to create lists more efficiently than when using classic procedures, such as e.g. using a for loop to incrementally add elements to a list.

It is often used as a filter to select items that meet a condition.

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

In square brackets we present the expression on the basis of which the new element is formed

- the creation of a new element is repeated based on the elements iterated in the **for** section
- the iteration can be supplemented with a condition

Creating a list based on elements defined by a loop:

```
# put i in the list, which you obtain as an element of the
loop iteration for range(10)
my_list = [i for i in range(10)]
print(my_list)
```

**Program output:**

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
#put the triple i in the list, which you get as an element of
the loop iteration for range(10)
my_list = [i*3 for i in range(10)]
print(my_list)
```

**Program output:**

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

## 5.4.2

Put together an entry that produces multiples of 4 starting at 12 and ending at 40.

```
my_list = _____ 4 _____
range(_____,11) _____
print(my_list)
```

- (
- while
- ]
- i
- [
- i
- +
- for
- \*
- )
- in
- 3

## 5.4.3

### List iteration

However, more often than creating a list, we process elements of an already existing one:

For the list on the input, create a list containing twice the values of this list.

```
my_list = [1,5,8,7,2]
new = [i*2 for i in my_list]
print(new)
```

**Program output:**

```
[2, 10, 16, 14, 4]
```

We get the new list by traversing the elements of the list, which ensures

```
for i in my_list
```

Before the iterative part, the form of the new element is defined by calculation or its modification:

```
i*2
```

The fact that these are elements of a list is expressed by placing the created sequence of created elements in square brackets. This list is inserted into the variable `new`.

```
new = [i*2 for i in my_list]
```

#### 5.4.4

Complete the code that will create a new list based on the values of the original list so that it contains the squares of the original elements:

```
my_list = [1,5,8,7,2]
new = [x * _____ for _____ _____ my_list]
print(new)
```

#### 5.4.5

### Working with text

Just as we work with numeric elements, we can also work with text values. We can use the functions for working with text as we like.

**From an existing list of strings, create a new one in which each element begins with an uppercase letter.**

```
my_list = ['anna', 'beata', 'cynthia', 'daniel']
new = [name[0].upper()+name[1:] for name in my_list]
print(new)
```

**Program output:**

```
['Anna', 'Beata', 'Cynthia', 'Daniel']
```



We iterate through the list so that the corresponding element appears under the **name** variable:

```
for name in my_list
```

We define the new element as

```
name[0].upper()+name[1:]
```

that is, we change the first character (index 0) of the string to a capital letter and the rest of the string, i.e. from the first character to the end of the string is added to this first letter.

### 5.4.6

Complete a program that creates a new list by taking the first three characters from each element and adding to them the original number of characters in the string:

```
my_list = ['anna', 'beata', 'cynthia', 'daniel']
new = [item[____:____] + '-' + ____ (len(item)) ____ item
in ____]
print(new)
```

Result:

```
['ann-4', 'bea-5', 'cyn-7', 'dan-6']
```

### 5.4.7

#### Filter

We often create a new string from an old one by omitting some values. In this case, we can add a condition to the end of the entry that decides whether the given element is added to the new list or not:

```
new = [expression for element in my_list if condition]
```

**From the existing list of numbers, include in the new one only those that are divisible by 7.**

```
my_list = [10,12,14,21,8,7,31,35]
new = [i for i in my_list if i % 7 == 0]
print(new)
```

**Program output:**

```
[14, 21, 7, 35]
```

- we iterate through the list, the iterating element is called **i**

- the element of the new list is also `i`, we don't change it
- we select only those elements in the new list where the condition `i % 7 == 0` is fulfilled

### 5.4.8

Complete a program that creates a new list by selecting male names (not ending in 'a') from the original list and correcting their first letter to capital:

```
my_list = ['anna', 'beata', 'cynthia', 'daniel', 'eva',
           'filip']
new = [name[____].____() + name[1:] for name ____ my_list
       if name[____] ____ 'a' ]
print(new)
```

Result:

```
['Daniel', 'Filip']
```

### 5.4.9

#### Fast transformation of input to numbers

Many times we encountered a situation where we needed to transform a list of input data obtained, for example, using `split`, into numbers. We usually followed each element of the list.

```
data = input('Numbers divided by spaces').split()
numbers = []
for num in data:
    numbers.append(int(num))
print(numbers)
```

**Program output:**

```
Numbers divided by spaces 1 2 5 8 77 9 1 [1, 2, 5, 8, 77, 9, 1]
```

The **map** function in Python applies a given function to each item in an iterable (like a list). When transforming a list of strings to integers, `map` helps by applying the `int` function to each string in the list.

```
string_numbers = '1 2 5 8 77 9 1'.split()
integer_numbers = list(map(int, string_numbers))
print(integer_numbers)
```

**Program output:**

```
[1, 2, 5, 8, 77, 9, 1]
```

- Function `int()` is the function that converts a string to an integer.
- Iterable `string_numbers` is the list of strings that we want to transform.

- map execution: `map(int, string_numbers)` applies `int` to each item in `string_numbers`: `int("10") → 10`, `int("20") → 20`, `int("30") → 30`, `int("40") → 40`
- The map function returns a `map` object, which is an iterator. Converting it to a list with `list()` gives: `[1, 2, 5, 8, 77, 9, 1]`

### 5.4.10

Complete the code to quickly transform the input into a list of numbers

```
string_numbers = input()._____()
_____ = _____(_____ (_____, _____))
print(integer_numbers)
```

- string\_numbers
- int
- map
- list
- split
- integer\_numbers

## 5.5 List (Programs II.)

### 5.5.1 The most common occurrence

The input is a list of names separated by spaces. Write the name that appears the most times in the list. If there are more such names, list them all in the order in which they first appeared in the list.

```
input: Adam Eva Peter Michael Laura Peter Eva
output:
Eva
Peter
```

### 5.5.2 Unique values

The input is a list of names separated by spaces. Print the names that appear in the list so that each name is printed only once. List them in the order in which they appeared in the list. Write them in the form of a list.

```
input: Ivan Jan Michal Jozef Jozef Jozef
output: [Ivan, Jan, Michal, Jozef]
```

### 5.5.3 Unique values

The input is a list of names separated by spaces. Print the names that appear only once in the list. Print them in the order in which they appeared in the list. Print them in the form of a list.

```
input: Ivan Jan Michal Jozef Jozef Jozef
output: [Ivan, Jan, Michal]
```

### 5.5.4 Above average values

The input contains a sequence of integer values separated by spaces - both positive and negative. Calculate and write the average rounded to two decimal places in the form: average: 1.51.

Then create and print the list of the elements that are above average (i.e. greater than the average) in the order corresponding to the original order on the input.

```
input:
5 1 6 3 2 7 8 4
output:
average: 4.00
above average: [5, 6, 7, 8]
```

### 5.5.5 Merging lists

The input is two lists, each on a separate line, with integer values separated by spaces. Create a third list so that each value is included only once, and the elements in the new list are ordered by size.

```
input:
3 3 2 1 3 3 3
5 8 4 4 4 4 4
output:
1 2 3 4 5 8
```

### 5.5.6 Name control

The input is a list of names separated by spaces. Make sure to create a new list so that all names start and end with a capital letter. At the same time, arrange the elements alphabetically. Print the resulting list in the form of a list. Use the comprehension sheet.

```
input: adam EVA peter michael Laura
output: ['AdaM', 'EvA', 'LaurA', 'MichaeL', 'PeteR']
```

### 5.5.7 Absolute values

The input contains a sequence of integer values separated by spaces - both positive and negative. Use list comprehension to create a list containing the absolute values of the original numbers, but only include numbers that are not divisible by three. Then print the resulting list in the form of a list

```
input:  
5 1 6 3 -2 7 -8 4  
output:  
[5, 1, 2, 7, 8, 4]
```

# Lists and Memory

Chapter **6**

## 6.1 List in memory

### 6.1.1

#### List in memory

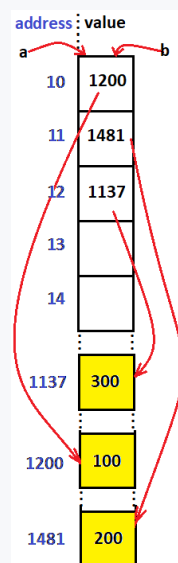
We create a list in memory, e.g. by listing the elements:

```
a = [100, 200, 300, "car", 50.04]
```

For each element, the necessary memory space is allocated - in this case 5 spaces, but in a different range (capacity)

The values of the elements are not stored in the list, but each element of the list is stored in "random" locations in memory, and the list contains only the addresses of these random locations.

In a simplified way, the situation can be imagined as follows:



If we make an assignment:

```
b = a
```

... it means that the same list in memory is referenced by two variables (as in the picture)

Overwriting an element at a specific location in a list appears to overwrite that element in both lists.

It actually gets overwritten in a single place, but by having both lists refer to the same place it looks like both are changed.

```
b[1] = 0
a[2] = 0
print(a)
print(b)
```

**Program output:**

```
[100, 0, 0, 'car', 50.04]
[100, 0, 0, 'car', 50.04]
```

Such behavior is typical for lists or more complex data structures.

In the case of simple variables of standard types, assigning values between variables just copies them.

### 6.1.2

Which statements are true?

- when creating a list, more space is reserved than its elements take up individually
- the contents of the list are not stored in a contiguous block of memory
- the contents of the list are stored in a contiguous memory block
- only the memory addresses of the elements are stored in the list
- the list stores the elements, not their memory addresses

### 6.1.3

What will be stored in list **b** after the following commands are executed?

```
a = [100, 200, 300, 400, 500]
a[3] = 10
b = a
b[2] = 7
a[1] = b[4]
b = [_____, _____, _____, _____, _____]
```

- 10
- 300
- 100
- 10
- 200
- 500
- 500
- 400
- 7
- 17
- 7



 6.1.4

## Adding an element

When adding a new element to a list, we usually take two approaches:

- merging lists

```
a = [1, 2, 3]
a = a + [4]
print(a)
```

**Program output:**

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

- adding via the **append()** command

```
a = [1, 2, 3]
a.append('new')
print(a)
```

**Program output:**

```
[1, 2, 3, 'new']
```

 6.1.5

Select the correct notations to add elements to the list **a**:

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

- a.append(4)
- a.append('4')
- a += [4]
- a = a + [4]
- a += 4
- a = a + 4
- a = a + (4)

 6.1.6

## Behavior when adding elements

A list at the level of a memory address list is created in memory in a contiguous block for fast access to elements.

If we add to the existing list in the following way:

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

```
b[1] = 'edited'
a.append('new')
print('a:',a)
print('b:',b)
```

**Program output:**

```
a: [1, 'edited', 3, 'new']
b: [1, 'edited', 3, 'new']
```

Using the **append()** command maintains a continuous block, meaning that the content of **b** is also consistent with **a**.

If the preallocated memory is exhausted, it creates a new contiguous block and copies the original elements into it.

The minimum number of elements in a block is around 200 when first used.

### 6.1.7

Choose the correct statements:

- using the `append()` command does not break the block contiguity for the addresses in the list
- using the `append()` command breaks the block contiguity for the addresses in the list
- using the `append()` command does not respect block contiguity for the data in the list
- using the `append()` command maintains block contiguity for the data in the list

### 6.1.8

In this program:

```
a = [1, 2, 3]
b = a
a = a + ['new']
print('a:',a)
print('b:',b)
```

**Program output:**

```
a: [1, 2, 3, 'new']
b: [1, 2, 3]
```

a new variable is created in the 4th line.

This is because the assignment command itself works by creating a new variable.

A new variable is created by concatenating the contents of the original list and the list with one element.

This step breaks the concatenation of lists **a** and **b**.

### 6.1.9

What will lists **a** and **b** contain after executing the following commands?

```
a = [10, 20, 30]
b = a
a = a + [40]
b.append(50)
```

```
a: [_____, _____, _____, _____]
b: [_____, _____, _____, _____]
```

- 20
- 40
- 40
- 10
- 50
- 10
- 20
- 30
- 30
- 50

### 6.1.10

What will lists **a** and **b** contain after executing the following commands?

```
a = [10, 20, 30, 40, 50]
b = a
a[3] = 10
b[4] = 80
b.append(60)
a = a + [70]
a.append(90)
b = b + [0]
```

```
a: [10, 20, _____, _____, _____, _____, _____, _____]
b: [10, 20, _____, _____, _____, _____, _____]
```

- 0
- 20
- 70

- 10
- 0
- 60
- 70
- 90
- 30
- 30
- 20
- 80
- 60
- 90
- 10
- 80

## 6.2 List elements

### 6.2.1

#### Behavior in basic operations

Using slice is based on returning a list as the result of the operation, so it doesn't change the original list.

```
a = [1, 2, 3, 4, 5]
b = a[1:3]
print('b:',b)
print('a:',a)
```

**Program output:**

```
b: [2, 3]
a: [1, 2, 3, 4, 5]
```

### 6.2.2

Which elements does list b contain?

```
a = list(range(1,21,3))
b = a[3:8:2]
b: [_____, _____]
```

- 13
- 19
- 6
- 14
- 4

- 9
- 21
- 2
- 20
- 15
- 8
- 17
- 7
- 12
- 11
- 5
- 16
- 1
- 18
- 10
- 3

### 6.2.3

The list comprehension also returns a new list as a result of its activity:

```
a = [1, 2, 3, 4, 5]
b = [i for i in a if i % 2 == 0]
print('b:',b)
print('a:',a)
```

**Program output:**

```
b: [2, 4]
a: [1, 2, 3, 4, 5]
```

### 6.2.4

What will list b contain after the program ends?

```
a = [1, 2, 3, 1, 6, 2]
b = [i*i for i in a[1:] if i < 6]
b: [_____, _____, _____, _____]
```

- 8
- 6
- 3
- 7
- 2
- 1
- 5
- 2
- 5
- 9

- 3
- 4
- 8
- 9
- 4
- 7
- 1
- 6

## 6.2.5

### Assignment to slice

In addition to retrieving a slice from a list, we can modify the list by assigning a list of elements to the slice, e.g.:

```
my_list = [1, 2, 3, 4, 5]
my_list[1:3] = [11, 13]
print(my_list)
```

**Program output:**

```
[1, 11, 13, 4, 5]
```

In this case, two elements have been replaced by two new elements.

Especially in the Python language, it is also possible to implement an operation that replaces some number of elements with a completely different number of elements. However, such operations are supported, apparently due to the orientation of the Python language for data processing - such an operation is not supported in "classic" programming languages.

E.g.:

```
my_list = [1, 2, 3, 4, 5]
my_list[1:3] = [99] # two elements are replaced by one
print(my_list)
```

**Program output:**

```
[1, 99, 4, 5]
```

It is even possible to replace a smaller number of elements with a larger one:

```
my_list = [1, 2, 3, 4, 5]
my_list[1:3] = [0, 0, 0, 0]
print(my_list)
```

**Program output:**

```
[1, 0, 0, 0, 0, 4, 5]
```

 6.2.6

What will be the result of the following assignment?

```
my_list = [1, 2, 3, 4, 5]
my_list[4:5] = [0, 1, 0, 1]
print(my_list)
my_list: [_____, _____, _____, _____, _____, _____, _____,
          _____]
```

- 0
- 1
- 1
- 1
- 1
- 2
- 0
- 4
- 3
- 1
- 0
- 0

 6.2.7

### Comparing lists

Lists are compared in the same way as strings - it goes through the elements of the list and when the first difference is found, the elements are compared:

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 5, 4]
print(a == b)
print(a < b)
```

**Program output:**

```
False
True
```

If an existing element is compared to a non-existent one, it is logical that the existing one is larger:

```
a = [1, 2]
b = [1, 2, 3, 5, 4]
print(a < b)
```

**Program output:**

```
True
```

Similarly for strings:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
b = ['alphabet', 'ate', 'girl', 'said', 'on', 'bear']
print(a < b)
```

**Program output:**

```
False
```

However, the problem is the combination of text and numeric value - these cannot be compared:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
b = ['alphabet', 1, 2, 3, 5, 4]
print(a < b)
```

**Program output:**

```
TypeError
```

```
'<' not supported between instances of 'str' and 'int'
```

## 6.2.8

Select the pairs for which the result of the comparison **a < b** is **True**:

- a: [1, 2], b: [1, 2, 3]
- a: [1, 2, 3, 1, 2, 3], b: [1, 2, 3, 4]
- a: ['mama', 'has', 'Ema'], b: ['mama', 'has', 'adam']
- a: [1, 2, 3, 4, 5], b: [1, 2, 3, 4]
- a: [0], b: [-1, -2, -3]
- a: ['mama', 'has', 'Dana'], b: ['mama', 'has', 'Adam']

## 6.2.9

### Join

The function that allows us to retrieve and process the elements of a list given as a text string is **split()**. The function splits the text based on the specified delimiter - often a space, comma or comma + space. Its result is the creation of a list:

```
string = "Mother has very small Ema."
my_list = string.split()
numbers = "1, 2, 3, 6, 99"
my_list2 = numbers.split(', ')
print(my_list)
print(my_list2)
```

**Program output:**

```
['Mother', 'has', 'very', 'small', 'Ema.']
['1', '2', '3', '6', '99']
```



The opposite operation is provided by the `join()` function, which creates a text string from the list with the separator we enter.

The `join()` function is a text string method of the form:

```
my_list = ['1', '2', '3', '5']
separator = ' '
result = separator.join(my_list)
print(result)
```

**Program output:**

```
1 2 3 5
```

The condition, of course, is that the elements of the list are in the form of text strings.

For example for the loaded input, we can easily change its form using separators:

```
data = input().split()
print(data)
text = ';' .join(data)
print(text)
text = '--' .join(data)
print(text)
```

**Program output:**

```
1 2 3 4 5 6 ['1', '2', '3', '4', '5', '6']
1;2;3;4;5;6
1--2--3--4--5--6
```

### 6.2.10

Complete the code so that you get even values from the interval **a-b** connected by three dots: for example, for 1-10 it will be:

```
2...4...6...8...10
```

```
a = int(input())
b = int(input())
z = [_____(i) for i in _____(a, b+1) if i _____ 2 == _____]
string = _____._____ (_____)
print(string)
```

- str
- join
- ''
- //
- string
- z

- %
- 0
- range
- '...'
- int

## 6.3 Arrangement

### 6.3.1

#### Arrangement

Let's go back to arranging lists once more.

The `sorted()` method returns a new list containing the result of sorting:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
print('a before:', a)
b = sorted(a)
print('a sorted:', b)
print('a after:', a)
```

**Program output:**

```
a before: ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
a sorted: ['alphabet', 'ate', 'bear', 'grandpa', 'on', 'said']
a after: ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
```

The `sort()` function - rearranges the elements => the list changes:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
print('a before:', a)
a.sort()
print('a after:', a)
```

**Program output:**

```
a before: ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
a after: ['alphabet', 'ate', 'bear', 'grandpa', 'on', 'said']
```

Again, operations between incompatible types are not allowed:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear', 4]
print('a before:', a)
a.sort()
print('a after:', a)
```

**Program output:**

```
a before: ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear', 4]
TypeError
```

```
'<' not supported between instances of 'int' and 'str'
```

### 6.3.2

What will be the result of the following operation?

```
a = ['alphabet', 'ate', 'Grandpa', 'Said', 'on', 'bear']
b = sorted(a)
b: ['_____', '_____', '_____', '_____', '_____', '_____']
```

- on
- ate
- bear
- Grandpa
- Said
- alphabet

### 6.3.3

#### Arrangement parameters

The function `sorted()` has two optional parameters - **reverse** and **key**

- **reverse** allows you to reverse the sort order

```
my_list = [1, 2, 8, 4, 5]
my_list2 = sorted(my_list, reverse = True)
print(my_list2)
```

**Program output:**

```
[8, 5, 4, 2, 1]
```

- **key** allows you to define rules for ordering through a function

For example to arrange the values based on the absolute value, we use:

```
my_list = [-1, 2, 8, 4, -5]
my_list2 = sorted(my_list, key = abs)
print(my_list2)
```

**Program output:**

```
[-1, 2, 4, -5, 8]
```

The values of the list will not change, but this function will be applied to them before they are compared.

So the comparison step will not have the form of `2 > -5`, but `abs(2) > abs(-5)`

### 6.3.4

How the numbers will be arranged after applying the following setting:

```
my_list = [1.3, -2.1, -8.05, 4.5, 5.4]
my_list2 = sorted(my_list, key = round, reverse = True)
z2: [_____, _____, _____, _____, _____]
```

- 5.4
- 4.5
- -8.05
- -2.1
- 1.3

### 6.3.5

It is also possible to define a custom function as a function for arrangement. The assumption is that it will have one parameter and return one value.

It can also be operations for which there is no function as such, e.g. the square of a number, which sorts the numbers by their square:

```
def square2(a):
    return a * a

my_list = [0, -1, -5, 2, 7]
my_list2 = sorted(my_list, key = square2)
print(my_list)
```

**Program output:**

```
[0, -1, -5, 2, 7]
```

or ordering by the size of the decimal part:

```
import math
def des_cast(a):
    return a - math.floor(a)

my_list = [1.3, -2.1, -8.05, 4.5, 5.4]
my_list2 = sorted(my_list, key = des_cast)
print(my_list2)
```

**Program output:**

```
[1.3, 5.4, 4.5, -2.1, -8.05]
```

In this case, it should be noted that for negative numbers, the decimal part is calculated to the nearest smaller number, so for -2.3 it is 0.7 to -3.

### 6.3.6

What will be the result of the following program, how will be the **l2** arranged?

```
import math
def des(a):
    return a - math.floor(a)

my_list = [1.3, 2.1, 8.05, 4.5, 5.4]
l2 = sorted(my_list, key = des)
l2: [_____, _____, _____, _____, _____]
```

- 2.1
- 8.05
- 1.3
- 4.5
- 5.4

### 6.3.7

When working with strings, it is common to sort strings regardless of character size by using the **lower** function from the **str** package, e.g.:

```
a = ['alphabet', 'Ate', 'grandpa', 'Said', 'on', 'Bear']
b = sorted(a)
print(b)
c = sorted(a, key = str.lower)
print(c)
```

**Program output:**

```
['Ate', 'Bear', 'Said', 'alphabet', 'grandpa', 'on']
['alphabet', 'Ate', 'Bear', 'grandpa', 'on', 'Said']
```

### 6.3.8

Complete the code to arrange the list according to the length of its strings so that the longer strings are at the beginning:

```
a = ['alphabet', 'ate', 'grandpa', 'said', 'on', 'bear']
b = _____(a, _____ = _____, _____ = True)
print(b)
```

- reverse
- reversed
- oposite
- len
- length

- value
- sorted
- key
- length
- id
- sort

## 6.4 List as a parameter

### 6.4.1

#### Function parameter

We know that:

- the parameter acts as a variable whose task is to "bring" a value to the function
- by using a variable in a function, we are actually working with the value it represents
- the value that is inserted into the variable is specified when the function is called

```
def my_sum(a, b):  
    return a + b  
  
s = my_sum(10, 20)  
print(s)  
x = 10  
y = 5  
z = my_sum(x, y)  
print(z)
```

**Program output:**

```
30  
15
```

### 6.4.2

What will be the value of the variable a after the termination of the program?

```
def my_sum(a, b):  
    amount = a + b  
    return amount
```

```
amount = 30
```

```
a = 10
b = 5
z = my_sum(a, b)
print(amount)
```

### 6.4.3

#### List as a parameter

We can pass the list to the function just like other parameters of different data types.

We can search it or process individual elements.

A program for finding the sum of elements in a list can take the form:

```
def my_sum(z):
    s = sum(z)
    return s

my_list = [1, 2, 3, 4, 5]
z = my_sum(my_list)
print(z)
```

**Program output:**

```
15
```

### 6.4.4

Complete the program that finds the number of list elements that contain the specified character:

```
def number(character, my_list):
    number = 0
    for i _____ my_list:
        _____:
            if i.index(_____) _____ 0:
                number = number + 1
            _____:
                continue
    _____ poc

my_list = ['mama', 'has', 'very', 'small', 'emil']
z = number('o', my_list)
print(z)
```

## 6.4.5

### Editing list in function

When parameters are changed in a function, the changes are not normally reflected outside the function.

However, if the content of the variable representing the list is changed, this change will also **be reflected** in the original variable.

E.g.:

```
def my_sum(z):
    s = sum(z)
    z.append(s) # we will also add sum to the end of the list
    return s

my_list = [1, 2, 3, 4, 5]
print('original:', my_list)
z = my_sum(my_list)
print('new:', my_list)
```

#### Program output:

```
original: [1, 2, 3, 4, 5]
new: [1, 2, 3, 4, 5, 15]
```

This behavior is due to the fact that the list is passed to the function not as a copy of the value, but as a **memory reference**.

This is due to the slowness of copying the contents of the list, the potential use of large amounts of memory, and consistency with the behavior of lists in other programming languages.

The consequence is that changes to the contents of the list in the function also change its contents in memory, which remain in the changed form even after the function is exited.

We can use this behavior very effectively.

Attention, the use of the list as a variable into which the value is inserted in this case does not mean that it is a local variable that will disappear after the end of the function. It is true that a change implemented in any way in a variable of type list is reflected in the memory of the list:

```
def process(my_llist):
    my_llist.append("second")
    my_llist += ["third word"]
```



```
my_list = ["first"]
process(my_list)
print(my_list)
```

**Program output:**

```
['first', 'second', 'third word']
```

#### 6.4.6

What does the following program print?

```
def process(list1):
    list1.append("word")
    list1.append(max(list1))
    list1.append(len(list1))
    return list1

my_list = input().split()
number = len(my_list)
process(my_list)
print(len(my_list) - number)
```

#### 6.4.7

What does the following program print?

```
def process(list1):
    list1.append("word")
    list1+= [max(list1)]
    list1+= [min(list1)]
    list1+= [len(list1)]
    return list1

my_list = input().split()
number = len(my_list)
process(my_list)
print(len(my_list) - number)
```

#### 6.4.8

We process the list in the method by modifying its values and the changes are reflected in the original source.

Let's take a program in which we edit names so that they start with an uppercase letter and other characters are lowercase.

We will not use letter comprehension in this case, even if it is offered:

```
def process(list1):
    for index, item in enumerate(list1):
        list1[index] = item[0].upper() + item[1:].lower()

my_list = ["anna", "beTa", "ceCILIA", "daNA"]
process(my_list)
print(my_list)
```

**Program output:**

```
['Anna', 'Beta', 'Cecilia', 'Dana']
```

Pitfalls we must face:

- we make changes in the list, not in the item variable
- to manipulate the list we need the index of the item
- **enumerate** is more elegant than **range**

### 6.4.9

What does the following program print:

```
def process(list1):
    for index, item in enumerate(list1):
        if item % 2 == 0:
            item = -item

my_list = [1, 2, 3, 4, 6]
process(my_list)
print(my_list[1] + my_list[3])
```

### 6.4.10

#### Creating a list

Often we need to create an empty list filled with some initial value. Thanks to direct access to the list, we can provide this directly in the universal method.

The input to it can be the number of elements and the value set in the list as a starting point.

```
def create_list(number, default = 0):
    return [default] * number

l1 = create_list(5, 1)
print(l1)
l2 = create_list(5)
print(l2)
```

```
l3 = create_list(4, '')
print(l3)
l4 = create_list(5, 'nothing')
print(l4)
```

**Program output:**

```
[1, 1, 1, 1, 1]
[0, 0, 0, 0, 0]
['', '', '', '']
['nothing', 'nothing', 'nothing', 'nothing', 'nothing']
```

### 6.4.11

Complete the function activity results:

```
def create_list(number, default = 0):
    return [default * number]

l1 = create_list(5, 1)
print(l1)                # prints [_____]
l2 = create_list(5)
print(l2)                # prints [_____]
l3 = create_list(4, '')
print(l3)                # prints [_____]
l4 = create_list(3, 'nothing')
print(l4)                # prints [_____]

```

### 6.4.12

#### Editing the list

When preparing general methods for data processing, it is necessary to decide whether the goal is to change the values of the list.

The decision depends on the situation, the scope of the list and the goals of further processing.

The following pair of functions perform the same operation with different effects on the original list:

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

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

```

```

original = [1, 3, 8, 5, 6]
new = add1(original, 'x1')
original.append('x2')
print('original:', original)
print('new:', new)
print('-----')

dependent = add2(original, 'y1')
original.append('y2')
print('original:', original)
print('dependent:', dependent)

```

**Program output:**

```

original: [1, 3, 8, 5, 6, 'x2']
new: [1, 3, 8, 5, 6, 'x1']
-----
original: [1, 3, 8, 5, 6, 'x2', 'y1', 'y2']
dependent: [1, 3, 8, 5, 6, 'x2', 'y1', 'y2']

```

The **add1()** function does not change the original content, as a result it returns a new (independent) list.

The **add2()** function modifies the list (the original list is changed) and at the same time returns the result (a memory pointer to the changed list).

 **6.4.13**

What elements will the new one contain after performing the following operations?

```

def o1(my_list):
    return my_list[0:-1]

def o2(my_list):
    my_list.append(1)
    return my_list

def o3(my_list):
    return my_list + [2]

list1 = [1, 3, 8, 5, 6]
a = o1(list1)
a.append(5)
b = o2(list1)
b.append(4)
c = o3(list1)
c.append(6)

```

```
print(list1)
List: [1, 3, 8, _____, _____, _____, _____]
```

- 2
- 6
- 4
- 5
- 6
- 1
- 5

## 6.5 Programs

### 6.5.1 Editing the list

Write a program that reads a space-separated list of integers (both positive and negative) from input and modifies them by zeroing negative values and doubling positive ones. Perform the operation in the function.

```
input:
1 -1 2 3 -5 4
output
2 0 4 6 0 8
```

### 6.5.2 Generator

Write a function that reads the number of elements and the default value from the input and returns a list with the specified number of elements filled with the specified value.

```
input: 10 a
output: ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

```
input: 10 -1
output: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

**file1.py**

```
def prepare(n, value):
```

```
# main
```

```
print(prepare()) # edit line
```

### 6.5.3 Unrepeatable

Write a program that, from an input containing a list of values, discards those that appear more than once. Thus, only the values that were originally in it only once will remain in the list. Solve this by using the function in which you edit the list:

```
input: 1 2 3 5 4 2 1
output: 3 5 4
```

```
input: mama has food has food
output: mama
```

### 6.5.4 Intersection

The input contains two lists of elements separated by spaces. Lists are placed on separate lines. Select the elements from the first list that also appear in the second list. Keep the original order of the elements.

```
input:
1 2 3 4
1 5 8 7 4 3
output:
1 3 4
```

### 6.5.5 Mutual exchanges

The input is a list of values separated by spaces. On the next lines are the pairs of elements to be exchanged in the list. The list of pairs ends with the value -1. Write a program that reads this data, performs element swaps and prints the resulting list.

```
input:
1 2 3 4 5 6
1 3
0 5
-1
output:
6 4 3 2 5 1
```

### 6.5.6 Division

The input is a list of values separated by spaces. The next line contains the number of sublists into which the original list should be divided. If the number of elements is less than the number of sublists, let the text "cannot be" be printed. If the list can be divided, divide it so that there is the same number of elements in each sub-list, only in the last one the number that remained and their number is less than in the previous ones.

```

input:
1 2 3 4 5 6 7
4
output:
1 2
3 4
5 6
7

```

### 6.5.7 Rotation

The input is a list of values separated by spaces. On the next line there is a value determining how many elements the list should be rotated by. It can be positive (elements rotate to the right) or negative (elements rotate to the left). Write a rotated list.

```

input:
1 2 3 4 5 6 7
3
output:
5 6 7 1 2 3 4

```

```

input:
1 2 3 4 5 6 7
-2
output:
3 4 5 6 7 1 2

```

### 6.5.8 Selection from connection

The input is three lists of values separated by spaces. The lists are stored below each other. The fourth line contains a numerical value. Write a program that creates a list of unique elements from these three lists but greater than the given value. Let the list be ordered from the largest to the smallest number.

```

input:
1 2 3 4 5 6
2 3 4 5 6
8 9 7 4 5 6 1 2 3
4
output:
9 8 7 6 5

```

### 6.5.9 Subsequence

Write a program that finds the longest sequence of identical elements in a list. The input is a list of values separated by spaces, the output is the repeated value, the number of repetitions and the starting index.

```
input: 1 2 2 2 3 3 4 5 6 3 3 3 3 2 3
output: 3, 7, 4
input: 1 2 2 2 3 3 3
output: 2, 3, 1
```

### 6.5.10 Subsequence II.

Write a program that finds the longest sequence of identical elements in a list. The input is a list of values separated by spaces, the output is the repeated value, the number of repetitions and the starting index. If there are several lists of the same length, print them all in the order of occurrence of the value in the original field.

```
input: 1 2 2 2 3 3 4 5 6 3 3 3 2 3
output:
2, 3, 1
3, 3, 9
```

```
input: 1 2 2 2 3 3 3
output:
2, 3, 1
3, 3, 4
```



# Tuple

## Chapter 7

## 7.1 Tuple

### 7.1.1

#### Mutable a immutable

Lists, or collections specifically in Python are generally divided into:

- **mutable**
- **immutable**

A typical mutable structure is a **list**, its non-editable version is a **tuple**. It supports practically all operations as in a list, just without operations that change the contents of the list.

What's the point of using another data structure? Wouldn't a list structure be enough for us? Advantages of tuple over list:

- Speed - a tuple is by default faster than a list because it is immutable and has a simpler structure.
- Less memory consumption - a tuple does not have as many built-in methods as a list, which means that it is usually less memory intensive and has less overhead.
- Simultaneous (parallel) assignment - tuple allows assignment of multiple values, which is useful, for example, for returning multiple values from a function at once.

### 7.1.2

Complete correctly:

list - \_\_\_\_table, \_\_\_\_ memory requirement, \_\_\_\_ speed

tuple - \_\_\_\_table, \_\_\_\_ memory requirement, \_\_\_\_ speed

- bigger
- greater
- smaller
- lower
- immu
- mu

### 7.1.3

Why do we need immutable elements? After all, they represent lists without the possibility of changing the content of elements

Benefits again:

- faster list iteration, more efficient work with memory, faster copying
- are **usually** used to store data of different types (name, surname, date of birth, salary), while lists are for similar/same
- they provide the programmer with the assurance that the data will not be overwritten - some parts of the values should not be changed: e.g. the name and surname representing the person should not allow changing only the first name or only the surname, if a change is made, then the entire record
- there is no need to synchronize them during multi-threaded code
- more advanced programming techniques
- they copy the security standards of other languages

#### 7.1.4

### Tuple

A tuple represents a tuple produced from any iterable sequence. As with a list, it is an ordered collection of elements. We define tuples using parentheses:

```
point = (100, -75)
print(point)
print(type(point))
```

**Program output:**

```
(100, -75)
```

Unlike lists, tuples are immutable = once a tuple is created, it cannot be changed in any way.

In the case of the point mentioned above, it gives some logic - changing some coordinate represents a completely different point - it should not be possible to change only one of the coordinates.

If we want to change a point, we delete it and create a new one.

#### 7.1.5

Complete the notation of the three tuples that represent the sides of the triangle:

```
triangle = _____ 100, 75, 25 _____
```

#### 7.1.6

### Creating a tuple

Creating an empty tuple is a dubious operation, since a tuple variable cannot be changed, and therefore its contents cannot be set/changed.

```
a = ()
x = tuple()
print(type(a), a)
print(type(x), x)
```

Program output:

```
()
()
```

It only makes sense to fill the variable with values - for a tuple, we use round brackets to define the object type:

```
pointA = (10, 10)
pointB = (10, 10, 10)
```

A special case is a one-element set, which we write with a comma after the value:

```
value = (10,)
```

The reason is that the entry...:

```
value = (10)
```

...is evaluated as the mathematical notation of the value in parentheses and thus in this case as an **int**

```
print(type(value))
```

Program output:

### 7.1.7

Select notations that represent tuples:

- a = (20,)
- a = (20, 30)
- a = (20, 30, 40)
- a = ()
- a = (20)
- a = (20, 30,)

### 7.1.8

#### Elementary operations

- connecting with "+"
- chaining with "\*"

```
t1 = (10, 20, 30)
t2 = (1, 2, 3)
t3 = t1 + t2
print(type(t3), t3)
t4 = t1 * 3
print(type(t4), t4)
```

**Program output:**

```
(10, 20, 30, 1, 2, 3)
(10, 20, 30, 10, 20, 30, 10, 20, 30)
```

### 7.1.9

What is the result of the following operation?

```
t1 = (10,)
t2 = (1, 2)
t3 = t1 * 3 + 2 * t2 + (t2 + t1)
print(t3)
```

### 7.1.10

## Operations with tuple

The basic operations correspond to those already learned from strings and lists:

- **len()** - number of elements
- **in** - checks whether the element is in the list
- **count()** - number of occurrences
- **index()** - position of first occurrence

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(len(t))
```

**Program output:**

```
11
```

```
if 2 in t:
    print('i have')
else:
    print('i do not have')
```

**Program output:**

```
i have
```

```
print('value 3', t.count(3), 'x')
```

**Program output:**

```

value 3 3 x
searched_for = 6
try:
    i = t.index(searched_for)
    print(searched_for,'on the position',i)
except:
    print('not found')

```

**Program output:**

```
not found
```

### 7.1.11

Add the correct functions to the following operations with tuples:

\_\_\_\_\_ - number of elements

\_\_\_\_\_ - checks if the element is in the list

\_\_\_\_\_ - number of occurrences

\_\_\_\_\_ - position of first occurrence

- in
- sum()
- exist()
- index()
- len()
- count()
- min()

## 7.2 Elements manipulation

### 7.2.1

#### Access to the element

The tuple element is accessed via index:

```

t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(t[1])

```

**Program output:**

```
2
```

Attempting to change the content of an element results in an error:

```
t[1] = 7
```

Program output:

```
TypeError
```

```
'tuple' object does not support item assignment
```

### 7.2.2

Fill in the results to the following code:

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 1, 7, 16, 6)
print(t[3])           # prints: _____
print(t.count(1))     # prints: _____
print(t.count('3'))   # prints: _____
print(len(t))         # prints: _____
print(0 in t)         # prints: _____
```

- 3
- False
- 2
- Key Error
- 12
- 5
- 1
- 0
- 4
- True
- Error
- 2
- 0

### 7.2.3

Support for functions working on a list is identical to functions for a list. But not in the form of methods, but functions:

- **sum()** - sum of elements
- **min()** - minimal element
- **max()** - maximum element

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(sum(t))
print(min(t))
print(max(t))
```

Program output:

```
54
```

```
1
```

```
11
```

### 7.2.4

Complete the correct syntax to find the results of the following operations:

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(_____)      # sum of elements
print(_____)      # minimal element
print(_____)      # maximum element
```

### 7.2.5

A slice in a tuple works in the standard way, its result is a tuple - it also does not support editing:

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(t[1:4])
x = t[7:8]
print(x)
```

**Program output:**

```
(2, 3, 4)
(8,)
```

Attention, when reading the element at the specified position, the result is not a tuple:

```
x = t[7]
print(x, type(x))
```

**Program output:**

```
8
```

### 7.2.6

The content of which variables obtained from variable t can be edited?

```
t = (1, 2, 3, 4, 3, 5, 'data', 8.6, 11, 3, 5)
```

- a = t[0]
- a = t[6]
- a = t[7]
- a = t[2:5]
- a = t[3:4]
- a = t[6:7]



## 7.2.7

### Tuple iteration

By default, we iterate through the for cycle, while the element has a type corresponding to the form in which it was inserted:

```
t = (1, '2', 3.5, "word", 3, True, 9)
for element in t:
    print(element, type(element))
```

**Program output:**

```
1
2
3.5
word
3
True
9
```

## 7.2.8

Complete the program that loops through the list of elements in a tuple and creates a second tuple list with the string lengths of the first tuple.

```
t = ('attention', 'bad', 'dog', 'bites', 'even', 'trough',
    'mouthpiece')
lengths = _____
for element _____ t:
    lengths = _____ (element) _____
print(lengths)
```

- (
- in
- +
- ()
- )
- lengths
- len
- ,

## 7.2.9

### Tuple transformation

Despite the fact that we create and populate the list in order to process it as efficiently as possible, sometimes it is necessary to change the type from immutable to mutable during the operation:

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
l1 = list(t)
print(l1)
l1.remove(2)
print(l1)
```

**Program output:**

```
[1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5]
[1, 3, 4, 3, 5, 9, 8, 11, 3, 5]
```

Note that the following notation creates a single tuple element as part of a mutable list:

```
l2 = [t]
print(l2)
```

**Program output:**

```
[(1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)]
```

## 7.2.10

Select the operations that are allowed in a program starting as follows:

```
tu = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
li = list(tu)
```

- tu = tu + (1,)
- li.remove(8)
- print(tu[2])
- print(li[2])
- li[1:3] = [8, 3]
- tu.append(8,)
- tu = tu + (1)
- tu.remove(8)
- tu[1:3] = (8, 2)

## 7.3 Use of tuple

### 7.3.1

#### Conversion to tuple

Again, there are also functions where it is necessary to convert another data type to a tuple. Similar rules apply to lists:

```
t = tuple('Python')
print(t)
```

**Program output:**

```
('P', 'y', 't', 'h', 'o', 'n')
```

```
t = tuple([2, 3, 5, 7])
print(t)
```

**Program output:**

```
(2, 3, 5, 7)
```

```
t = tuple(range(1, 11))
print(t)
```

**Program output:**

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

### 7.3.2

Complete the code that creates a tuple from the string:

```
t = _____('Python')
print(t)
```

```
result: ('P', 'y', 't', 'h', 'o', 'n')
```

### 7.3.3

#### Multiple assignment

We can write the simultaneous assignment of several values to several variables in one line:

```
a, b, c = 1, 2, 3
print(a, b, c)
```

**Program output:**

```
1 2 3
```

### 7.3.4

Fill in the correct values for variables **a** and **b**:

```
a, b = 7, 5
a, b = a + b, a - b
a, b = 3 + a - b, a * b
print(a)          # prints: _____
print(b)          # prints: _____
```

### 7.3.5

#### Returning multiple outputs from a function

Python allows multiple values to be returned from a function thanks to tuples. A tuple in this case is taken as a single element consisting of practically any number of other elements.

An example can be e.g. simultaneously returning the contents and circuit of the rectangle:

```
def rect_info(a, b):
    ct = a * b
    cc = 2 * a + 2 * b
    return ct, cc

content, circuit = rect_info(5, 6)
print(content, circuit)
```

**Program output:**

```
30 22
```

If we load the values into one variable, a variable of the type is created in this case, which is in accordance with the return value of the function, that is, we have an immutable object of the tuple type available.

With this result, we can perform all operations allowed for tuples:

```
result = rect_info(2, 4)
print(result)
print(type(result))
```

**Program output:**

```
(8, 12)
```

### 7.3.6

Assign the correct data type to the variables:

```
def rect_info(a, b):
    x = a * b
    y = 2 * a + 2 * b
    return x, y

content, circuit = rect_info(5, 6)
result = rect_info(5, 6)
```

```
content - type: _____
circuit - type: _____
result - type: _____
x - type: _____
y - type: _____
```

- float
- float
- list
- float
- list
- int
- tuple
- list
- int
- tuple
- int
- int
- tuple
- list
- tuple
- float

### 7.3.7

**Find all the divisors of the given number and write whether it is a prime number.**

- task is a nice example of a combination of getting a tuple and processing it further
- in the function we first create a list of divisors
- then we write them out and if their number is 2, it is a prime number, otherwise it is a composite number

```
def divisors(n):
    d = ()
    for i in range(1, n + 1):
        if n % i == 0:
            d = d + (i,)
    return d
```

```
x = 7 # int(input())
my_list = divisors(x)
print(my_list)
# print(", ".join(my_list))
if len(my_list) == 2:
    print("prime number")
else:
    print("composite number")
```

**Program output:**

```
(1, 7)
```

```
prime number
```

## 7.4 Variadic function

### 7.4.1

#### Variadic function

A variadic function is a function that accepts a variable number of arguments. It means that this function can be called with different number of input values.

We already know a certain form of a function with a different number of parameters, it is a function that has some parameters set to initial values.

If we do not specify these parameters when calling the function, the default ones will be used.

```
def add(a, b = 0, c = 0, d = 0):
    return a + b + c + d
```

If we enter one parameter, the result will be the entered value.

If we enter two, they are added and the default values - 0 are used instead of c and d.

```
print(add(10))
print(add(20, 30))
```

**Program output:**

```
10
50
```

We could also use 3 or 4 inputs, but with 5 parameters it is no longer possible to match the value to the corresponding variable and the result is an error.

```
print(add(1, 2, 3, 4, 5))
```

**Program output:**

```
TypeError
```

```
add() takes from 1 to 4 positional arguments but 5 were given
```

### 7.4.2

What is the name of a function that accepts a variable number of arguments?

- variadic
- multiconceptual
- multiparametric

### 7.4.3

#### Wrapped parameter

If we need to send an unknown and arbitrary number of parameters to the function, we use the so-called wrapped parameter.

It can be distinguished from other parameters by the fact that it is preceded by the character `*`.

This parameter allows any number of parameters to be obtained from the function call and iterated in the function.

E.g.:

```
def add(*numbers):
    s = 0
    for i in numbers:
        s = s + i
    return s
```

```
sm = add(1,2,3)
print(sm)
```

**Program output:**

```
6
```

What type is the **number** variable in the function?

```
def add(*numbers):
    print(type(numbers))
    s = 0
    for i in numbers:
        s = s + i
```

```

return s

sm = add(1,2,3)
print(sm)

```

**Program output:**

```

6

```

It is a **tuple!!!**

### 7.4.4

Complete the variadic function designed to multiply any number of numbers.

```

def product(____ numbers):
    pro = ____
    for ____ numbers:
        pro = pro ____ i
    return pro

```

### 7.4.5

**Write a function that, for a given list, returns a given number of random values from it.**

In this case, the first parameter will define the number of elements for the resulting tuple, followed by an unknown number of elements to choose from.

The wrapped parameter is listed last in order to avoid potentially misidentifying its values and other values that do not belong to it.

```

import random
def choose(l, *my_list):
    elements = ()
    for i in range(l):
        elements = elements +
(my_list[random.randrange(len(my_list))],)
    return elements

print(choose(3, 1,2,3,4,5,6,7))
print(choose(3, 'a','b','c','d','e','f','g'))
print(choose(3, 'a','b'))

```

**Program output:**

```

(4, 4, 1)
('e', 'd', 'd')
('a', 'a', 'a')

```



### 7.4.6

Complete the code that finds the sum of the first n elements of the list. Provide functionality for both numeric and text types. Note the data type validation method:

```
def amount(_____, _____my_list):
    if type(my_list[0]) is int:
        sum = _____

    if type(my_list[0]) _____ _____:
        sum = ''

    for i in range(z):
        sum = sum + _____[_____]
    return _____

print(amount(3, 1,2,3,4,5,6))
print(amount(3, 'a', 'b', 'c'))
```

### 7.4.7

If we choose the list generated by **range()** as the list of values intended for the variadic function, a small problem arises:

```
import random
def choose(n, *my_list):
    elements = ()
    for i in range(n):
        elements = elements +
(my_list[random.randrange(len(my_list))],)
    return elements

print(choose(3, range(8)))
print(type(range(8)))
```

**Program output:**

```
(range(0, 8), range(0, 8), range(0, 8))
```

Range creates a special element passed to the function as an element of type **range**.

In order to send the generated values instead, we have to "unpack" it - again with **"\*"**.

```
print(choose(3, *range(8)))
```

**Program output:**

```
(6, 0, 3)
```

We can also (and will need to) expand the **list** before sending it to the variadic function. We send one element of type **list** without whitewashing.

```
print(choose(3, *['Adam', 'Beta', 'Cecil', 'Dana']))
```

**Program output:**

```
('Beta', 'Adam', 'Dana')
```

 7.4.8

Which records send to the **select(n, \*z)** function a list of 7 elements: 0,1,2,3,4,5,6.

- choose(7,0,1,2,3,4,5,6)
- choose(7, \*range(7))
- choose(7, range(7))
- choose(0,1,2,3,4,5,6)
- vyber(0,1,2,3,4,5,6,7)

 7.4.9

**Print?**

The **print()** function is a typical variadic function that allows the output of any number of elements of virtually any type:

```
print(range(8))
print(*range(8))
print(['Eva', 'Fedor', 'Gusto', 'Hana'])
print(*['Eva', 'Fedor', 'Gusto', 'Hana'])
```

**Program output:**

```
range(0, 8)
0 1 2 3 4 5 6 7
['Eva', 'Fedor', 'Gusto', 'Hana']
Eva Fedor Gusto Hana
```

## 7.5 Programs

 7.5.1 Powers

The input is a space-separated list of numbers. Create a tuple containing the squares of the given numbers and print it.

```
input: 1 2 5 -1
```

```
output: (1, 4, 25, 1)
```

### 7.5.2 Filter

The input is a list of numbers separated by spaces. The next line contains an integer value. Create and print tuples containing a list of numbers from the input that are greater than the specified value.

```
input:
5 8 7 2 1 3 5
3
output:
(5, 8, 7, 5)
```

### 7.5.3 Interval filter

The input is a list of numbers separated by spaces. In the next line there are two values representing a closed interval. Create and print a tuple containing a list of numbers from the input that are in the specified interval.

```
input:
5 8 7 2 1 3 9
3 7
output:
(5, 7, 2, 1, 3)
```

### 7.5.4 Chain

Write a program that uses the variadic function to combine words separated by commas on the input. Let the function return one string, which will be the union of all input strings, while the # character will be used as a separator in the returned string.

```
input: mother,father,kid,dog,cat
output: mother#father#kid#dog#cat
```

```
input: mother
output: mama
```

### 7.5.5 Punctuation

For a given input, write a program that prints a list of punctuation marks from the given input. Let the data be stored in tuple() and arranged in the order in which they appear in the text, but each character only once.

You can get the list of punctuation characters from the *string* module, where the constant is available:

```
string.punctuation = r"!\"#$%&-'()*+,-
./:;<=>?@[^_`{|}~\""
```

```
input : Hello world! (1+1=2)
output: ('!', '(', '+', '=', ')')
```

```
input : Order: 100$ + 10%; teddy@gmail.com
output: (':', '$', '+', '%', ';', '@', '.')
```

```
input : priscilla
output: ()
```

### 7.5.6 Number of characters, lines, sentences and words

Write the code that detects how many characters, rows, sentences and words are contained in the specified text file. The name of the text file is given at the input. Suppose words do not divide at the end of a line, and no sentence ends with three dots. Print the following information to the console: "characters: 67 rows: 2 sentences: 9 words: 14".

```
Input : book.txt
Output: characters: 70 rows: 3 sentences: 5 words: 16
```

```
Input : book2.txt
Output: characters: 34 rows: 2 sentences: 2 words: 7
```

Preview of text file book.txt:

```
Hello, how are you? I'm fine. And you?
This test is a test.
It tests itself!
```

**file1.py**

```
public class JavaApp {

    public static void main(String[] args) {
        // write your code here

    }
}
```

### 7.5.7 Median and mode

Write a program that, for numeric values separated by commas at the input, calculates their median and mode in a separate function, while returning both values at once.

- The median is the middle value after ordering the values.
- Mode is the most common value.

```
input: 1,3,1,5,4
output: (3,1)
```

```
input: 5,7,1,1,1,2,3,5,5,7,7,7,7,7
output: (5,7)
```

### 7.5.8 Navigation

Imagine that you are the captain of a ship in a 2D space represented by a Cartesian coordinate system.

The position of the rocket is defined by two coordinates (x and y). Your mission always starts at the origin of the coordinate system (where  $x == 0$  and  $y == 0$ ).

The plan of each mission is coded with signs representing the cardinal points and numbers representing the number of steps to be taken in that direction. Example of a mission plan list:

```
mission_plan = ['S',3,'Z',2,'J',1]
```

The rocket starts at position 0.0 and then moves 3 times north (to position 0.3). Next, the rocket moves 2 times to the west (to position -2,3). Finally, the rocket moves south and reaches the target position -2.2.

**Fix the work-in-progress solution** to work correctly according to the following requirements:

- The function must be variadic, i.e. j. able to accept a different number of parameters (always an even number, according to the examples below).
- The function must return a tuple containing the coordinates of the rocket after the mission is completed and the direct distance from the starting position (in this case the result must be written to 2 decimal places).

```
input : S,3,Z,2,J,1
output: -2, 2, 2.83
```

```
input : S,1,Z,1,J,10,V,2
output: 1, -9, 9.06
```

**start.py**

```
import math
# fix the following solution:
# plan must be a wrapped parameter, not a list!
def go_to_mission(plan):
    x = 0
    y = 0
    for i in range(0, len(plan), 2):
        if plan[i] == 'S':
            y += plan[i + 1]
        elif plan[i] == 'J':
            y -= plan[i + 1]
        elif plan[i] == 'V':
            x += plan[i + 1]
        else:
            x -= plan[i + 1]
    d = math.sqrt(x * x + y * y)

# load the mission plan
data = input()

# call the function, extract the actual parameters from the
data list
pos_x, pos_y, distance = go_to_mission()

# list the 3 values of the returned tuple
print(f'{}, {}, {:.2f}')
```

 **7.5.9 Calculation of absence**

Write the code that finds the average number of absences in the specified text file. At the input, is given the file name that contains the student name in each row and a colon-separated number of absence hours. Print the number of registered pupils, the total and the average number of absences on the console. Round the number to one decimal place.

```
input : data1.txt
output:
10
122
12.2
```

Preview of text file data1.txt:

```
Anna:12
```

```
Jano:10  
Peter:20  
Adam:30  
Mato:5  
Jozo:15  
Fero:16  
Miro:4  
Jana:7  
Dana:3
```

**file1.py**

```
public class JavaApp {  
    public static void main(String[] args) {  
        // write your code here  
  
    }  
}
```

# List of Lists

Chapter **8**



## 8.1 Matrix introduction

### 8.1.1

Imagine you have data about students' scores in three subjects: Math, Science, and English. You could store this data in three separate lists:

```
math_scores = [85, 90, 78]
science_scores = [88, 92, 84]
english_scores = [82, 89, 80]
```

A separate loop is required to find the average from each subject without using functions:

```
math_sum = 0
for i in math_scores:
    math_sum += i
print('math', math_sum / len(math_scores))
science_sum = 0
for i in science_scores:
    science_sum += i
print('science', science_sum / len(science_scores))
english_sum = 0
for i in english_scores:
    english_sum += i
print('science', english_sum / len(english_scores))
```

**Program output:**

```
math 84.33333333333333
science 88.0
science 83.66666666666667
```

If you want to group all this information together for easier access and better organization, you can use a **matrix**, which is a list of lists.

```
scores = [
    [85, 90, 78], # Math scores
    [88, 92, 84], # Science scores
    [82, 89, 80] # English scores
]

print(scores)
```

**Program output:**

```
[[85, 90, 78], [88, 92, 84], [82, 89, 80]]
```

Now, the scores are stored in a single structure, and you can access or process them more efficiently.

- A matrix stores related data in rows and columns.
- Each list inside the matrix is like a "row" in a table.

### 8.1.2

Add the necessary characters to the code

```
scores = _____ [85, 90, 78 _____, _____ 88, 92, 84 _____] [82, 89, 80 _____]
```

- ),
- (
- )
- [
- ]
- ]
- ],
- ]
- ],
- )
- ]
- )
- [
- (

### 8.1.3

A matrix is a collection of numbers organized into rows and columns, like a table. In programming, matrices are often represented as *lists of lists*. Each inner list is a row, and the elements in the inner lists are the entries in the columns. Here is a 2x3 matrix (2 rows and 3 columns):

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]
```

- Rows are represented by horizontal lines of numbers (e.g., [1, 2, 3] is the first row).
- Columns represent vertical lines of numbers (e.g., 1 and 4 are in the first column).

### 8.1.4

Create a 3x2 matrix where the first row contains 10, 20, the second row 50, 60, and the third row 40, 30.

```
[ [_____, _____],
  [_____, _____],
  [_____, _____] ]
```

- 50
- 10
- 30
- 40
- 60
- 20

### 8.1.5

#### Accessing elements in a matrix

You can access specific elements in a matrix using two indices:

- the row index.
- the column index.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]

print(matrix[0][1]) # Output: 2 (first row, second column)
print(matrix[1][2]) # Output: 6 (second row, third column)
```

Indices start at **0** as usually in Python. To access an element is used **matrix[row][column]**.

### 8.1.6

What value is stored in the second row, first column of this matrix:

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

- 4
- 6
- 3

### 8.1.7

When working with data, it's common to have lists inside lists. For example, let's say we have a list of coordinates:

```
coordinates = [[1, 2], [3, 4], [5, 6]]
```

Here, each inner list represents a point in 2D space (x, y). Using this matrix-like structure, you can organize and process large datasets easily.

#### So, why use matrices?

- Compact representation - instead of creating multiple lists, you use one matrix.
- Logical grouping - each row can represent an object, and each column can store attributes of that object.

```
coordinates = [
    [1, 2], # Point 1: (1, 2)
    [3, 4], # Point 2: (3, 4)
    [5, 6]  # Point 3: (5, 6)
]

for point in coordinates:
    print(f"Point: x = {point[0]}, y = {point[1]}")
```

#### Program output:

```
Point: x = 1, y = 2
Point: x = 3, y = 4
Point: x = 5, y = 6
```

### 8.1.8

Create a matrix to store these three points in 3D space: (2, 9, 7), (1, 8, 4), (3, 0, 6)

```
_____' _____' _____
_____' _____' _____
_____' _____' _____
```

- 0
- 3
- 2
- 1
- 6
- 9
- 8
- 7
- 4

### 8.1.9

#### Searching the matrix

Nested loops are great for working with matrices. The outer loop goes through rows, and the inner loop goes through columns.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]

for row in matrix:
    for element in row:
        print(element, end=" ")
    print() # Move to the next line after printing a row
```

#### Program output:

```
1 2 3
4 5 6
```

This code shows how to iterate through a matrix and access all its elements row by row.

- The outer loop goes through each row in the matrix. An array is basically a list of lists, where each inner list represents a row. The loop selects one row at a time from the matrix.
- The inner loop goes through the elements of the current row, which it receives as an element (**row**) from the entire matrix, i.e. one sheet.

### 8.1.10

Complete the code for listing the elements of the matrix.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]

for row in ____:
    for ____ in ____:
        print(element ____ ____)
```

- end=" "
- sep=" "
- ,

- element
- print()
- row
- end
- matrix

## 8.2 Tables

### 8.2.1

A matrix is often thought of as a table where rows and columns can hold various types of data. While numbers are common, Python allows you to create matrices with mixed data types, such as strings, numbers, and even tuples.

Here's a matrix with different data types:

```
table = [
    ["Name", "Age", "Grade"],           # Row 1: Strings
    ["Alice", 20, 85.5],               # Row 2: String, Integer,
Float
    ["Bob", 21, 92.0],                 # Row 3: String, Integer,
Float
    ["Charlie", 19, 88.5]              # Row 4: String, Integer,
Float
]

# Printing the table
for row in table:
    print(row)
```

**Program output:**

```
['Name', 'Age', 'Grade']
['Alice', 20, 85.5]
['Bob', 21, 92.0]
['Charlie', 19, 88.5]
```

### 8.2.2

Create a matrix representing three students with their names, ages, and favorite subjects.

```
my_table = _____
    ['Name', 'Age', 'Subject'],
    ['_____', 21, _____],
    [_____, _____, 'chemistry'],
```

```
['Charlie', 19, '_____']
]
```

- [
- 'math'
- 'Alice'
- history
- 20
- Bob

### 8.2.3

Thanks to this variability of the list in the matrix, we can also combine different types of values, e.g. text and number. Thanks to this, the introductory task will be greatly simplified.

```
math = [85, 90, 78]
science = [88, 92, 84]
english = [82, 89, 80]
```

we can describe as

```
scores = [
    ["math", 85, 90, 78],
    ["science", 88, 92, 84],
    ["english", 82, 89, 80]
]
```

```
print(scores)
```

**Program output:**

```
[['math', 85, 90, 78], ['science', 88, 92, 84], ['english',
82, 89, 80]]
```

To calculate the score, it is enough to go through each line of the matrix and write the name of the subject listed in the first item.

If we look at the data in the row, the first data is text and the rest are numbers. The easiest way to calculate the average is to count ken with elements that are numbers.

After the end of the calculation = finishing the line, we will print the result

```
for row in scores:
    my_sum = 0
    for element in row:
        if type(element) == int:
            my_sum += element
```

```
print(row[0], '-', my_sum / (len(row) - 1)) # without 0th
element
```

**Program output:**

```
math - 84.33333333333333
science - 88.0
english - 83.66666666666667
```

A special operation in the code is the expression

```
if itype(element) == int
```

which checks if the given element is a number.

### 8.2.4

Which code to check whether it is a positive whole number is correct?

- if type(element) == int:
- if type(element) == 'int':
- if type(element) == "<class 'int'>":

### 8.2.5

As long as we know in which columns what values are stored, we do not need to check the data type, but we will focus only on the considered columns.

- In that case, we will not go through the elements in the row in a cycle, but we will consider indexes.
- In the second cycle, we start from the second element (index 1) and end at the last one.

```
scores = [
    ["math", 85, 90, 78],
    ["science", 88, 92, 84],
    ["english", 82, 89, 80]
]
for row in scores:
    my_sum = 0
    for ind in range(1, len(row)):
        my_sum += row[ind]
    print(row[0], '-', my_sum / (len(row) - 1)) # without 0th
element
```

**Program output:**

```
math - 84.33333333333333
science - 88.0
english - 83.66666666666667
```



**range(1,len(row))** - generates values for the index, starting from 1 and ending at a position one less than the number of row elements. This exactly covers our needs - for 4 elements it will skip the element at the 0th position and generate the values 1,2,3.

### 8.2.6

How many values will the command **range(1,len(row))** generate:

```
row = [1,4,5,8,7,9,5,4,7,3,5]
```

- 10
- 9
- 11
- 12

### 8.2.7

Alternatively, if we also want to transform the first cycle into an index use, we can modify it:

```
scores = [
    ["math", 85, 90, 78],
    ["science", 88, 92, 84],
    ["english", 82, 89, 80]
]
for i in range(len(scores)):
    my_sum = 0
    for j in range(1,len(scores[i])):
        my_sum += scores[i][j]
    print(scores[i][0], '-', my_sum/(len(scores[i]) - 1))
```

- in a cycle with variable **i**, **range()** generates a range of 0 and the number of rows of the matrix
- in the loop with variable **j**, the **range()** from 1 to the number of elements in the i-th row is generated - **scores[i]**
- and finally, access to the elements is through the element on the i-th row and j-th column, which we write via **scores[i][j]**

### 8.2.8

Complete the code to print the value 10 from the matrix.

```
matrix = [['Name', 'Age', 'Grade'], ['Alice', 20, 'Fail'],
          ['Bob', 22, 'Pass'], ['Diana', 10, 'Pass']]
print(_____)
```

- ]
- [
- [
- 2
- matrix
- 5
- 3
- 3
- 3
- 4
- 4
- ]
- 1
- 2
- 1

## 8.3 Operations in a table

### 8.3.1

We can represent the matrix in several ways. In general, we need to work with a tuple of tuples or a list of lists. Of course, we can also use their different combinations.

```
m1 = [[1,2],[3,4]]
m2 = ((1,2),(3,4))
```

```
print(m1)
print(m2)
```

**Program output:**

```
[[1, 2], [3, 4]]
((1, 2), (3, 4))
```

With a tuple we gain speed, with a list we gain editability. If we work with a list of lists, it is possible to edit each cell. You can modify a cell in a matrix by accessing it using its row and column indices.

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]
# Updating Bob's age
```

```

table[2][1] = 22
# Printing updated table
for row in table:
    print(row)

```

**Program output:**

```

['Name', 'Age', 'Grade']
['Alice', 20, 85.5]
['Bob', 22, 92.0]
['Charlie', 19, 88.5]

```

### 8.3.2

Update the grade of "Alice" to 90.0.

```

table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]

# Updating
table[____][____] = 90.0

```

### 8.3.3

#### Replace row

We can also replace an entire row by assigning a new list to the row's index.

```

table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]

print(table)
print('-----')
# Replacing Charlie's row
table[3] = ["Charles", 20, 93]

# Printing updated table
for row in table:
    print(row)

```

**Program output:**

```

[['Name', 'Age', 'Grade'], ['Alice', 20, 85.5], ['Bob', 21,
92.0], ['Charlie', 19, 88.5]]
-----
['Name', 'Age', 'Grade']
['Alice', 20, 85.5]
['Bob', 21, 92.0]
['Charles', 20, 93, 50]

```

Similar to the list, we just assign the list to the row index to replace it. It is usually convenient that the length of the new line matches the others, but in general this is not a condition.

 **8.3.4**

What is the result stores in **list1** after the following code

```

list1 = [10, 20, 30, 40]
list2 = [50, 60]
list1[2:2] = list2

```

- [10, 20, 50, 60, 30, 40]
- [10, 20, 30, 40, 50, 60]
- [10, 20, 50, 60, 30]
- [50, 60, 10, 20, 30, 40]

 **8.3.5**

A Matrix in Python is usually represented as a list of lists. Each row of the matrix represents an internal list, so to add a new row, we can use the **append()** method, which adds an item (in this case, a row) to the end of the list.

Suppose we have the following matrix (a 3x3 matrix) and we want to add a new row, [10, 11, 12], to the matrix.

```

# Matrix before adding a new row
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
# New row to add
new_row = [10, 11, 12]
# Add the new row to the matrix
matrix.append(new_row)
# Print the updated matrix

```

```
print("Updated Matrix:")
for row in matrix:
    print(row)
```

**Program output:**

```
Updated Matrix:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[10, 11, 12]
```

- The **append()** method adds the new row to the end of the matrix.
- Each new row must have the same number of elements as the existing rows if you're working with a consistent table structure (though technically, Python lists can hold rows of different lengths).
- You can add as many rows as needed by using **append()** multiple times.

 **8.3.6**

Create a 5-line matrix where each line contains 3 numbers. You need to build the matrix by appending rows one at a time, using the **append()** command.

The final matrix should look like this:

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

- `matrix.append([1, 2, 5])`
- `matrix.append([1, 8, 3])`
- `matrix.append([4, 5, 9])`
- `matrix.append([1, 5, 6])`
- `matrix.append([7, 2, 9])`
- `matrix = []`

 **8.3.7**

A common operation is to add data to an existing matrix based on user input. If we have a matrix in the form where the first row contains the names of the columns, we can use these column names when generating the input question. This makes it easier for users to understand what kind of data they need to provide.

For example, suppose we have the following matrix representing a simple employee database:

```
matrix = [
    ["Name", "Age", "Department"], # Column headers
    ["Alice", 30, "HR"],
    ["Bob", 25, "IT"],
    ["Charlie", 35, "Marketing"]
]
```

We want to ask the user to input a new employee's data (Name, Age, Department) and then add that data as a new row to the matrix with following steps:

- extract the column names from the first row.
- use the column names to create clear input prompts for the user.
- ask the user to provide data for each column.
- append the new data as a row to the matrix.

```
# Extract column names from the first row
columns = matrix[0]
# Collect user input for each column
new_row = []
for column in columns:
    user_input = input(f"Enter {column}: ")
    new_row.append(user_input)
# Add the new row to the matrix
matrix.append(new_row)
# Print the updated matrix
print("\nUpdated Matrix:")
for row in matrix:
    print(row)
```

**Program output:**

```
Enter Name: JanEnter Age: 99Enter Department: IT
Updated Matrix:
['Name', 'Age', 'Department']
['Alice', 30, 'HR']
['Bob', 25, 'IT']
['Charlie', 35, 'Marketing']
['Jan', '99', 'IT']
```

The easiest way is to add a new row to the matrix as a whole.

### 8.3.8

What is the most often used command to read values from user (console)?

- input
- get
- read

- readln

### 8.3.9

#### Combining tables

We can add new data by not only appending rows but also combining two matrices.

```
table = [  
    ["Name", "Age", "Grade"],  
    ["Alice", 20, 85.5],  
    ["Bob", 21, 92.0],  
    ["Charlie", 19, 88.5]  
]  
# Another table of students  
new_table = [  
    ["Eve", 20, "Pass"],  
    ["Frank", 21, "Fail"]  
]  
# Combining the tables  
table.extend(new_table)  
# Printing combined table  
for row in table:  
    print(row)
```

#### Program output:

```
['Name', 'Age', 'Grade']  
['Alice', 20, 85.5]  
['Bob', 21, 92.0]  
['Charlie', 19, 88.5]  
['Eve', 20, 'Pass']  
['Frank', 21, 'Fail']
```

### 8.3.10

Which command can be used to join values from two tables?

- append
- extend
- extends
- join

## 8.4 Reading and processing data

### 8.4.1

We already know how to read the data for one row of the table and fill it, let's now create the whole table by loading the data from the input.

The easiest way is to load the data item by item, provided that we specify the number of rows and the number of columns in the input:

- First, we need to know how many rows and columns the table will have.
- We will loop through each row and ask the user to input values for each column.
- For each row, we will collect the data and add it to the table.

```
# Input number of rows
rows = int(input("Enter the number of rows: ")) # Ask the
user for the number of rows and convert the input to an
integer
columns = int(input("Enter the number of columns: ")) # Ask
the user for the number of columns and convert the input to an
integer
table = [] # Initialize an empty list to hold the table
(matrix)
# Input rows
for i in range(rows): # Loop over the number of rows
    new_row = [] # Initialize an empty list to represent a new
row
    for j in range(columns): # Loop over the number of columns
for each row
        # Ask the user to input the element for the specific row
and column (e.g., [1,1], [1,2], etc.)
        element = input(f"Enter [{i + 1},{j+1}]:")
        new_row.append(element) # Add the input element to the
current row
    table.append(new_row) # After all columns for the row are
entered, add the row to the table
# Printing the table
for row in table: # Loop through each row in the table
    print(row) # Print the row
```

**Program output:**

```
Enter the number of rows: 2Enter the number of rows: 2Enter
[1,1] 1Enter [1,2] 2Enter [2,1] 3Enter [2,2] 4['1', '2']
['3', '4']
```



- The user provides data for each column in each row.
- The number of rows and columns is flexible, based on user input.
- The table is stored as a matrix (list of lists) where each row is a list.

### 8.4.2

Which of the following describes the purpose of the line `new_row.append(element)`?

```
for i in range(rows):
    new_row = []
    for j in range(columns):
        element = input(f"Enter [{i + 1},{j+1}] ")
        new_row.append(element)
    table.append(new_row)
```

- It adds each individual element to the row as the user input.
- It adds the entire table to the new\_row list.
- It adds a new row to the table after collecting all the elements.
- It appends the column number to the row.

### 8.4.3

#### Inserting data with value separators

The easiest way for both the user and the code to insert data into a table is to allow the user to input the entire row at once. To make it easier for the user, we can use a value separator, such as a space or comma, to separate the individual values in each row. Once the user inputs the data, the program can split the row based on the separator and create separate values for each column.

We can either:

- Ask the user to input the number of rows at the start, then read rows one by one.
- Allow the user to enter rows continuously, stopping when they enter a special value (e.g., 0 to stop entering rows).

```
# Initialize an empty table
table = []
# Ask the user for the number of rows (optional)
rows = int(input("Enter the number of rows (or 0 to enter rows
until stop): "))
# Input rows based on the user's number of rows or until '0'
is entered
while rows != 0: # for i in range(rows):
    # Ask the user to enter a row of data separated by space
```

```

    row_input = input("Enter a row of data separated by space:
")
    # Split the row into values and add it as a new row in the
table
    table.append(row_input.split()) # Split by spaces
    # Decrement the number of rows left to enter (if rows is
not 0)
    rows -= 1
# Print the resulting table
for row in table:
    print(row)

```

- or do not define the number of rows before loading data

```

# Initialize an empty table
table = []
# Alternatively, enter rows until the user types '0'
while True:
    row_input = input("Enter a row of data (or '0' to stop):
")
    if row_input == '0':
        break
    table.append(row_input.split()) # Split by spaces
# Print the resulting table
for row in table:
    print(row)

```

#### 8.4.4

You are creating a program that allows the user to input entire rows of a table using a separator (e.g., space or comma) between the values. The program will split the input into separate values and populate the table accordingly.

Which of the following best describes how the program processes a row of data entered by the user?

- The program splits the row into individual values based on a separator (e.g., space or comma) and stores them as separate items in a list.
- The program stores the entire row as a single string without any modification.
- The program asks the user to input each column individually before storing the data.
- The program automatically guesses the values for the row if the user doesn't enter anything.

## 8.4.5

We are working with a table that stores students' names and their grades in Math, English, and Chemistry. Our task is to add two new columns:

- History grade - prompt the user to input the grade for History for each student.
- Average Grade - calculate the average grade for each student (including History) and add it as the last column.
- original table:

Name	Math	English	Chemistry
Alice	85	90	80
Bob	75	70	65
Charlie	95	85	90

- final table:

Name	Math	English	Chemistry	History	Average
Alice	85	90	80	88	85.75
Bob	75	70	65	78	72.00
Charlie	95	85	90	92	90.50

So we start with the given structure of the table (Name, Mathematics, English, Chemistry).

- In the first step, we read the mark from the history for each registered student from the user and insert it into the list representing the row.
- Next, we go through the rows and add the calculation of the average from the saved ratings.

```
# Initial table
table = [
    ["Alice", 85, 90, 80],
    ["Bob", 75, 70, 65],
    ["Charlie", 95, 85, 90]
]
# Adding History grades
print("Enter the History grades for each student:")
for row in table:
    history_grade = int(input(f"Enter History grade for {row[0]}: "))
    row.append(history_grade)
```

**Program output:**

```

Enter the History grades for each student:
Enter History grade for Alice: 80Enter History grade for Bob:
60Enter History grade for Charlie: 98
# Calculate and add average grade
for row in table:
    grades = row[1:] # Extract grades only (ignoring the
name)
    average = sum(grades) / len(grades) # Calculate average
    row.append(round(average, 2)) # Add average as the last
column
# Print the final table
print("\nFinal Table:")
print(["Name", "Math", "English", "Chemistry", "History",
"Average"])
for row in table:
    print(row)

```

**Program output:**

```

Final Table:
['Name', 'Math', 'English', 'Chemistry', 'History', 'Average']
['Alice', 85, 90, 80, 80, 83.75]
['Bob', 75, 70, 65, 60, 67.5]
['Charlie', 95, 85, 90, 98, 92.0]

```

 8.4.6

What is the correct way to add a new column (e.g., "History") to an existing table in Python?

- Append the new value to each row in the table.
- Add a new row with the column name at the top of the table.
- Use `append()` to add the column directly to the table.
- Replace an existing column with the new column.

 8.4.7**Delete**

We can delete rows or columns using the `del` statement for rows or list comprehensions for columns.

- Deleting a row:

```

table = [['Name', 'Age', 'Grade'],
        ['Alice', 20, 'Fail'],

```

```

        ['Bob', 22, 'Pass'],
        ['Charlie', 19, 'Pass']]
# Deleting Bob's row
del table[2]
for row in table:
    print(row)

```

**Program output:**

```

['Name', 'Age', 'Grade']
['Alice', 20, 'Fail']
['Charlie', 19, 'Pass']

```

Deleting a column must again be performed by deleting an item in each row separately:

```

table = [['Name', 'Age', 'Grade'],
        ['Alice', 20, 'Fail'],
        ['Bob', 22, 'Pass'],
        ['Charlie', 19, 'Pass']]
# Deleting the 'Age' column
for row in table:
    del row[1]
# Printing updated table
for row in table:
    print(row)

```

**Program output:**

```

['Name', 'Grade']
['Alice', 'Fail']
['Bob', 'Pass']
['Charlie', 'Pass']

```

### 8.4.8

How many rows will remain in the table, or what value is printed after executing the following commands?

```

table = [['Name', 'Age', 'Grade'],
        ['Alice', '20', 'Fail'],
        ['Bob', '22', 'Pass'],
        ['Charlie', '19', 'Pass']]
for row in table:
    if row[1] >= '20':
        del(row)

print(len(table))

```

- 4
- 3
- 2
- 1

### 8.4.9

#### Find

Searching the table is quite simple. If we want to calculate or display data matching the search criteria, it is necessary to identify the correct column and focus on the comparison. If a value is found, we can print the entire row.

Let's find in our matrix e.g. how many students are 20 years old, and let's name them.

In this case, we can implement the statement in one cycle with the calculation of values.

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5],
    ["Daniel", 20, 88.5]
]
# Initialize a counter to track rows with Age = 20
cnt = 0
# Loop through each row, excluding the header (starting from
table[1:])
for row in table[1:]:
    # Check if the 'Age' column (index 1) equals 20
    if row[1] == 20:
        # Increment the counter for each matching row
        cnt += 1
        # Print the matching row
        print(row)
```

#### Program output:

```
['Alice', 20, 85.5]
['Daniel', 20, 88.5]
```

### 8.4.10

What is the result of the following code:

```
table = [
```

```

["Name", "Age", "Grade"],
["Alice", 20, 85.5],
["Bob", 21, 92.0],
["Charlie", 19, 88.5],
["Daniel", 20, 88.5]
]
cnt = 0
for row in table[1:]:
    if row[1] > 18 or row[1] < 20:
        cnt +=1
print(cnt)

```

- 4
- 3
- 2
- 1
- 5

### 8.4.11

#### Sort

We can sort rows based on specific columns, such as sorting students by age or grade.

```

table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]
# Sorting rows by age (column index 1)
sorted_table = sorted(table[1:], key=lambda row: row[1])
sorted_table.insert(0, table[0]) # Add header back
# Printing sorted table
for row in sorted_table:
    print(row)

```

#### Program output:

```

['Name', 'Age', 'Grade']
['Charlie', 19, 88.5]
['Alice', 20, 85.5]
['Bob', 21, 92.0]

```

- **sorted()** sorts data and return it as a result, and **key=lambda** specifies which column to use.

- headers are often excluded from sorting by `table[1:]` and added back after by `insert(0,table[0])`

#### 8.4.12

In which index of matrix `table` will Bob's age be stored after arrangement?

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5] ,
    ["Daniel", 22, 88.5] ,
    ["Ester", 17, 88.5] ,
    ["Filip", 16, 88.5] ,
    ["Helen", 18, 88.5]]
sorted_table = sorted(table[1:], key=lambda row: row[1])
sorted_table.insert(0, table[0])
```

- [2,1]
- [3,1]
- [5,1]
- [6,1]
- [7,1]

#### 8.4.13

##### Filter

We can extract specific rows or columns based on conditions, like finding students with grades above 90.

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]
# Filtering students with grades above 90
filtered_rows = [row for row in table[1:] if row[2] > 90]
# Printing filtered rows
for row in filtered_rows:
    print(row)
```

##### Program output:

```
['Bob', 21, 92.0]
```



Usual way is to create new filtered list. Common approach is to use **list comprehensions** to filter rows and skip the headers.

#### 8.4.14

What is the result of the following code?

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
    ["Daniel", 20, 88.5]
]
filtered_rows = [row for row in table if row[1] >= 20]
print(len(filtered_rows))
```

- error
- 3
- 1
- 2

#### 8.4.15

Adjust for proper program functionality:

```
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
    ["Daniel", 20, 88.5]
]
filtered_rows = [row for row in table[_____] if row[1] >=
_____]
print(filtered_rows)
```

- 0:1
- '20'
- int()
- 1:
- 20
- 1:1
- :

## 8.5 Data in files

### 8.5.1

#### Writing data to file

Files are a way to store data persistently, allowing programs to read and process data without requiring it to be hardcoded. This is useful when working with large tables or matrices.

First, we need to prepare the data files and work together as best we can to store different types of values.

```
# Table to save
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]
with open("output.txt", "w") as file:
    for row in table:
        # Manually convert each element in the row to a string
        row_as_strings = [str(element) for element in row]
        # Join the elements with commas and write to the file
        file.write(",".join(row_as_strings) + "\n")
```

The result of this activity is a file with the following content:

```
with open("output.txt", "r") as file:
    for row in file:
        print(row, end="")
```

#### Program output:

```
Name, Age, Grade
Alice, 20, 85.5
Bob, 21, 92.0
Charlie, 19, 88.5
```

### 8.5.2

What does the expression `row_as_strings = [str(element) for element in row]` do?

- Converts each element in row to a string and stores the result as a new list.
- Joins all elements in row into a single string.
- Converts row into a string without modifying its elements.
- Converts only numeric elements in row to strings, leaving others unchanged.

### 8.5.3

#### Read from file

If we don't just want to read data from a file and simply process it, but we need to save it for more complex processing, we usually transform it into a sheet or a tuple. Rereading takes place line by line and we store data in the lists in the form required by the current assignment.

Now let's read the data separated by a comma and store them in a matrix:

```
# Preparing data
table = [
    ["Name", "Age", "Grade"],
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5]
]
with open("data.txt", "w") as file:
    for row in table:
        # Manually convert each element in the row to a string
        row_as_strings = [str(element) for element in row]
        # Join the elements with commas and write to the file
        file.write(",".join(row_as_strings) + "\n")
# Reading the file and converting it into a list of lists
table = []
with open("data.txt", "r") as file:
    for line in file:
        row = line.strip().split(",") # Split the line into
elements based on commas
        table.append(row)
# Displaying the resulting table
for row in table:
    print(row)
```

#### Program output:

```
['Name', 'Age', 'Grade']
['Alice', '20', '85.5']
['Bob', '21', '92.0']
['Charlie', '19', '88.5']
```

The **strip()** command is necessary to remove unnecessary characters, e.g. space and **\n** at the end of the line

### 8.5.4

Fill the code to read data from file to list:

```

table = []
with open("data.txt", "r") as file:
    for line in file:
        row = line.strip().split(",")
        table.append(row)
for row in table:
    print(row)

```

- w
- row
- add
- ad
- split
- strip
- insert
- a
- r
- append

### 8.5.5

#### Selecting data from a file

We can perform operations like filtering rows or calculating values after reading data from a file.

**View information about students who are 20 years old or older.**

```

table = [
    ["Alice", 20, 85.5],
    ["Bob", 21, 92.0],
    ["Charlie", 19, 88.5],
    ["Daniel", 18, 34.5]
]
with open("data.txt", "w") as file:
    for row in table:
        row_as_strings = [str(element) for element in row]
        file.write(",".join(row_as_strings) + "\n")

```

Already during data loading, we can evaluate which data should be displayed and which should not.

```

with open("data.txt", "r") as file:
    for line in file:
        row = line.strip().split(",")
        if int(row[1]) >= 20: # Check if Age (2nd column) is
greater than 20
            print(row)

```

**Program output:**

```
['Alice', '20', '85.5']
['Bob', '21', '92.0']
```

 8.5.6

Complete the code for writing data from the table matrix:

```
with open("data.txt", "w") as file:
    for row in table:
        row_as_strings = [_____(element) for element _____ row]
        file.write(","._____(row_as_strings) + "_____")
```

- str
- \n
- in
- int
- add
- \t
- ,
- insert
- join

 8.5.7**Read, process and write**

You can read a file, modify its contents, and save the new data into another file.

The file contains data about students and their results from individual subjects separated by commas. Read the data, find out the average rating and write a list of students sorted by average into a new file, which will contain only name and average.

```
input:
Adam,3,3,3
John,1,3,4,2
output
Jan,2.5
Adam,3
```

Let's go to prepare file **students.txt**.

```
# Preparing data
table = [ ["Adam", 3, 3, 3],
          ["Bob", 2, 1, 2, 2],
          ["Jan", 1, 3, 4, 4 ,2]]
```

```
with open("students.txt", "w") as file:
    for row in table:
        # Manually convert each element in the row to a string
        row_as_strings = [str(element) for element in row]
        # Join the elements with commas and write to the file
        file.write(",".join(row_as_strings) + "\n")
```

We will read the data line by line, convert the marks from each line into numbers and then calculate the average from them. We store the obtained value together with the name of the students in a tuple. After loading all students, we sort the data according to the second column and write it in a new file.

```
students = []
# Open the input file and read the data
with open("students.txt", "r") as input_file:
    for line in input_file:
        data = line.strip().split(",") # Split the line by commas
        name = data[0] # First element is the student's name
        grades = [int(grade) for grade in data[1:]] # Convert
grades to integers
        average = sum(grades) / len(grades) # Calculate average
        students.append((name, average)) # Add name and average
as a tuple
# Sort the list of students by average grade
students.sort(key=lambda x: x[1])
# Write the sorted data to the output file
with open("sorted_students.txt", "w") as output_file:
    for name, average in students:
        output_file.write(f"{name}, {average:.1f}\n")
```

- and check:

```
with open("sorted_students.txt", "r") as f:
    for line in f:
        print(line, end="")
```

#### Program output:

```
Bob, 1.8
Jan, 2.8
Adam, 3.0
```

### 8.5.8

How can you sort a list of student data by their second column (e.g., age or average grade)?

- `students.sort(key=lambda x: x[1])`

- `students.sort(key=lambda x: x[0])`
- `students.sort(lambda x: x[1])`
- `students.sort(by_column=1)`

## 8.5.9

### Text processing

Process a text file containing sentences and count how many times each word appears. Sort the found words according to the number of occurrences, in the case of the same number of occurrences, according to the alphabet. Do not write words that appear only once in the text. Clear the words from the characters like,.-

We divide the read line into words and look for them in the list of words. If the word is in the list, we increase the number of its occurrences, if not, we add it to the list with the number of occurrences of 1.

```
table = [ "Python is fun.", "Python makes data processing
easy.", "Learning Python is rewarding." ]
with open("data.txt", "w") as file:
    for row in table:
        file.write(row + "\n")
# Initialize a list to store words and their counts as tuples
word_counts = []
# Define a list of characters to remove from words
unwanted_chars = ",.-"
# Open and read the text file
with open("data.txt", "r") as file:
    for line in file:
        # Remove unwanted characters manually
        cleaned_line = line
        for char in unwanted_chars:
            cleaned_line = cleaned_line.replace(char, "")

        # Split the cleaned line into words
        words = cleaned_line.lower().split()

        # Process each word
        for word in words:
            # Check if the word is already in the list
            found = False
            for i, (existing_word, count) in
enumerate(word_counts):
                if existing_word == word:
                    # If found, increment the count
```

```

        word_counts[i] = (existing_word, count +
1)
        found = True
        break

    # If the word is not found, add it with count 1
    if not found:
        word_counts.append((word, 1))
# Filter out words that appear only once
filtered_words = [item for item in word_counts if item[1] > 1]
# Sort the list by count (descending) and then alphabetically
sorted_words = sorted(filtered_words, key=lambda x: (-x[1],
x[0]))
# Write the results to an output file
for word, count in sorted_words:
    print(f"{word}: {count}")

```

**Program output:**

```

python: 3
is: 2

```

 8.5.10

Complete the code so that it cleans up inappropriate characters:

```

unwanted_chars = ",.-"
with open("data.txt", "_____") as file:
    for line in file:
        for char in _____:
            line = _____._____ (_____, _____)
        print(line)

```

- w
- line
- unwanted\_chars
- r
- line
- append
- unwanted\_chars
- replace
- char
- ""



## 8.6 Matrix in math

### 8.6.1

A matrix in mathematics is a rectangular array of numbers arranged in rows and columns. It is a powerful tool used to represent and manipulate data, often used in mathematics, physics, computer science, and engineering.

Typical matrix:

--- ERROR ---

The **size** of a matrix is defined by the number of rows and columns.

The matrix **A** above is a 3x2 matrix (3 rows, 2 columns).

We know that matrices are often represented in Python as lists of lists:

```
A = [
  [1, 2],
  [3, 4],
  [5, 6]
]
```

### 8.6.2

What is the size of the following matrix:

```
A = [ [1, 2], [3, 4], [5, 6], [7, 8], [9, 0] ]
```

- 5 x 2
- 2 x 5
- 2 x 2
- 5 x 5
- 10

### 8.6.3

In a square matrix (a matrix with the same number of rows and columns), **diagonals** are specific sets of elements.

#### Main diagonal

- The **main diagonal** consists of the elements where the row and column indices are the same.
- It runs from the **top-left corner** to the **bottom-right corner** of the matrix.
- In a matrix A, the elements of the main diagonal are **A[i][i]**, where  $i=0,1,\dots,n-1$

--- ERROR ---

The elements of main diagonal are: 1, 5, 9

### 8.6.4

What is the sum of elements on main diagonal in the following matrix:

```
A=[[1,2,3,7],[4,0,5,6],[7,1,8,9],[3,2,1,0]]
```

- 9
- 15
- 16
- 11

### 8.6.5

Complete the code that lists the elements on the main diagonal:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
n = _____(matrix)
diagonal = []
for i in range(_____):
    diagonal.append(matrix[_____][_____])
print(diagonal)
```

- i-1
- size
- n
- i+1
- i-1
- i
- i
- n-1
- i+1
- len
- n+1

### 8.6.6

#### Other (or secondary) diagonal

- The **other diagonal**, also known as the **secondary diagonal**, consists of the elements where the sum of the row and column indices is equal to  $n-1$ .
- It runs from the **top-right corner** to the **bottom-left corner** of the matrix.
- In a matrix A, the elements of the other diagonal are  $A[i][n-1-i]$ , where  $i=0,1,\dots,n-1$ .

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The elements of the secondary diagonal are: 3, 5, 7

### 8.6.7

What is the product of the elements on the main diagonal in the following matrix:

```
A=[[1,2,3,7],[4,0,5,6],[7,1,8,9],[3,2,1,0]]
```

- 105
- 0
- 16
- 21

### 8.6.8

#### Adding matrices

You can add two matrices of the **same size** by adding their corresponding elements.

```
matrix1 = [
    [1, 2, 3],
    [4, 5, 6]
]
matrix2 = [
    [7, 8, 9],
    [10, 11, 12]
]
# [ 1 + 7,  2 + 8,  3 + 9]
# [ 4 + 10, 5 + 11, 6 + 12]

result = []
for i in range(len(matrix1)):
    row = []
    for j in range(len(matrix1[0])):
        row.append(matrix1[i][j] + matrix2[i][j])
    result.append(row)

print(result)
```

Program output:

```
[[8, 10, 12], [14, 16, 18]]
```

### 8.6.9

What is the sum of the elements [0,2] and [1,2] in the matrix you get as the sum of the following two matrices

```
matrix1 = [ [1, 2, 3], [4, 5, 6] ]
matrix2 = [ [7, 0, 3], [4, 1, 5] ]
```

- 17
- 20
- 8

### 8.6.10

#### Matrix multiplication

Matrix multiplication involves multiplying rows of the first matrix with columns of the second. It means that for the dimensions of the matrices, the number of rows of the first must be the same as the number of columns of the second. It is possible to multiply 2 x 5 and 5 x 4 matrices, but not 5 x 4 and 2 x 5.

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 5 \end{bmatrix} \cdot \begin{bmatrix} 6 & 7 & 9 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 6 + 2 \cdot 7 & 1 \cdot 7 + 2 \cdot 8 & 1 \cdot 9 + 2 \cdot 9 \\ 1 \cdot 6 + 3 \cdot 7 & 1 \cdot 7 + 3 \cdot 8 & 1 \cdot 9 + 3 \cdot 9 \\ 1 \cdot 6 + 4 \cdot 7 & 1 \cdot 7 + 4 \cdot 8 & 1 \cdot 9 + 4 \cdot 9 \\ 2 \cdot 6 + 5 \cdot 7 & 2 \cdot 7 + 5 \cdot 8 & 2 \cdot 9 + 5 \cdot 9 \end{bmatrix}$$

In the example, 4x2 and 2x3 matrices are multiplied, while the resulting matrix has a dimension of 4x3. To find the element in the i-th row and j-th column of the resulting matrix, we take the sum of the products of the i-th row of the first matrix and the j-th column of the second matrix.

The code that ensures the matrix product has the form:

```
A = [[1, 2, 4], [3, 4, 5]]
B = [[5, 6], [7, 8], [1, 2]]
result = [[0, 0], [0, 0]]
for i in range(len(A)):          # Row of A
    for j in range(len(B[0])):    # Column of B
        for k in range(len(B)):  # Column of A / Row of B
```

```

        result[i][j] += A[i][k] * B[k][j]
print(result)

```

**Program output:**

```
[[23, 30], [48, 60]]
```

### 8.6.11

Complete the matrix multiplication result:

```

A = [[1, 3], [2, 4]]
B = [[5, 7], [6, 8]]
A * B = [[ _____ , _____ ],
         [ _____ , _____ ]]

```

## 8.7 Matrix (programs)

### 8.7.1 Reset values below the main diagonal

Write the code that creates a matrix (3x3 size) from the integer values obtained at the input and resets all elements below the main diagonal. The given 9 numbers are separated by a space at the input. Print the modified matrix on the console. Allocate 4 spaces for each value for the matrix.

```

input : 44 -2 45 -29 35 14 0 50 -34
output:
 44  -2  45
  0  35  14
  0   0 -34

```

### 8.7.2 Mirror matrix

Write the code that prints a mirror image flipped along a vertical axis for a matrix of size  $n \times n$  containing 0 and 1. At the input, is given  $n$ , each array element separated by a space. Print a mirror image of the matrix on the console.

```

input : 4 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
output:
1 1 0 0
1 1 0 0
1 1 0 0
1 1 0 0

```

### 8.7.3 Apartment building

In the block of flats, each floor is divided into apartments. The administrator records the number of inhabitants for each apartment.

For the specified dimensions of the block of flats - the first represents the number of floors, the second the number of apartments within the floor, read the number of inhabitants in the next lines, so that in each line the number of inhabitants of all apartments on the given floor are separated by spaces.

Determine which apartment has the most residents.

If there are several flats with the same number of inhabitants in the block of flats, indicate the one that is located highest. If there are several such flats on the given floor, indicate the one that is furthest to the right.

```
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
```

### 8.7.4 Random attacks

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
```

## 8.8 Tables (programs)

### 8.8.1 Mirror

Write the code that loads numbers from the given file and mirrors them. At the input, is given the name of the file that contains the data in the form of numbers, and saves in a number array of 10 elements. Print numbers from last to first on the console.

```
input : myData1.txt
output:
10
9
8
7
6
5
4
3
2
1
```

Preview of text file myData1.txt:

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

file1.py

```
public class JavaApp {
    public static void main(String[] args) {
        // write your code here

    }
}
```

### 8.8.2 Sum of numbers in the string

Write the code that calculates the sum of integers occurring in the string.

```
input : We have 12 hens at home, 54 geese and 3 ducks.
output: 69
```

```
input : 12.3,8 9
output: 32
```

### 8.8.3 Ordered sublist

The input is a sequence of numbers separated by spaces with an unknown number of elements. Find the longest subsequence in which all elements are in ascending order, print its length and elements as a list. If there are more sequences with the same number of elements, print the first one in the sequence (the beginning of which has a smaller index)

```
input:
10 11 13 21 18 16 17 18 24 27 15
output:
5
[16, 17, 18, 24, 27]
```

### 8.8.4 The largest average of the subset

The input is a sequence of numbers separated by spaces with an unknown number of elements. Find the contiguous (elements are next to each other) subset (at least 2 elements) that has the largest average. Print its length, average rounded to one decimal place, and elements as a list. If there are more sequences with the same number of elements, list the first one in the sequence (the beginning of which has a smaller index)

```
input:
10 11 13 21 18 16 17 18 24 27 15
output:
2
25.5
[24, 27]
```



### 8.8.5 Palindromes

The input is a sequence of numbers separated by spaces with an unknown number of elements. Find in it all palindromes with a length of at least 3 elements. A palindrome is a subset that contains the same elements when written from front to back. When printing, proceed by ordering the sequences from largest to smallest - you leave the ordering to Python.

```
input:
10 11 13 21 18 16 17 16 18 24 27 15
output:
[18, 16, 17, 16, 18]
[16, 17, 16]
```

### 8.8.6 The smallest subset

The input contains two sequences of numbers separated by spaces with unknown numbers of elements. Find a contiguous subset of the first list that contains the smallest possible number of elements such that all elements from the second list occur in it. If the entered pair does not have a solution, print "error". If there are more subsequences with the same number of elements, print the first one in the sequence (the beginning of which has a smaller index).

```
input:
10 11 13 21 18 16 17 18 24 27 15
18 16 15
output:
[16, 17, 18, 24, 27, 15]
```

```
input:
10 11 13 21 18 16 17 18 24 27 15
22 16
output:
error
```

### 8.8.7 ChemLab

Only three researchers a day are admitted to the chemistry lab visitor list. From time to time there will be a request to find out if the specified person is in the laboratory. Usually it is not clear whether it is a first or last name or its code.

Write a program that will have 9 input lines with three data for each of the three researchers. Put these data into a list of triples.

On the last line there will be a string, which can be just part of the name.

They will list all the data containing the searched string in the form of a tuple.

If the search string is not in the matrix, it prints "No match".

```
input :
Adam
Mally
1996
Matthew
Great
1987
Joseph
Carrot
1998
Adam
output ('Adam', 'Mally', '1996')
```

```
input :
Adam
Mally
1996
Matthew
Great
1987
Joseph
Carrot
1998
John
output:
No match
```

### 8.8.8 Cardiac surgery

Write a program that finds the average weight of the patients of the cardiac surgery department in the given text file.

At the input, the name of the file is given, which represents one department and in each line contains the name of the patient and the weight separated by a colon.

List the number of registered patients in the given department, their total and average weight. Load the data into an appropriate data structure and round the average to one decimal place.

```
input : data1.txt
output:
10
```

```
1032
103.2
```

Preview of text file data1.txt:

```
Anna:112
Jano:110
Peter:87
Adam:130
Mato:56
Jozo:153
Fero:116
Miro:94
Jana:75
Dana:99
```

### 8.8.9 Visitors

The input contains the name of the file in which the data on the number of visits to the respective city is stored.

Write a program that reads data from a file and sorts it according to the column (0-2) specified in the next line.

It then lists them in the form as shown below:

```
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

**Set**

**Chapter 9**

## 9.1 What is set

### 9.1.1

A set is a built-in Python data structure used to store **unique and unordered items**. Unlike lists or tuples, sets do not allow duplicate values. They are often used when you need to eliminate duplicates or perform mathematical operations like unions and intersections.

- sets are defined using curly braces {}.
- sets automatically remove duplicates or does not add them to the list.
- there is no element position in the list - elements are stored in their own effective structure
- elements must be immutable, e.g. strings, numbers, tuples,
- it is not possible to work with a set of lists because lists are mutable structures
- adding an element is done using the **add()** command

```
# Creating a set
fruits = {"apple", "banana", "cherry"}
print(fruits)
# Adding
fruits.add("peanut")
# Adding a duplicate
fruits.add("apple") # No effect because "apple" is already in
the set
print(fruits)
```

**Program output:**

```
{'banana', 'cherry', 'apple'}
{'banana', 'cherry', 'apple', 'peanut'}
```

### 9.1.2

Fill in the code to add elements to the list

```
fruits = _____ "apple", "_____", "cherry"_____
fruits._____("mango")
print(fruits)
```

- insert
- (
- ]
- add
- banana
- }
- [

- )
- {
- append

### 9.1.3

Sets support several basic operations such as:

- adding elements - **add()**
- removing elements - **discard()** - given the absence of an index, we determine which element to delete
- removing elements - **remove()** - in the case of an attempt to remove a non-existent element, it generates an error
- the complete deletion of elements will be ensured by the **clear()** method
- number of elements - **len()**
- membership control - **in** returns True if element is in set

```
# Adding elements to a set
numbers = {1, 2, 3}
numbers.add(4)
print(numbers)  # {1, 2, 3, 4}

# Removing elements
numbers.remove(2)
print(numbers)  # {1, 3, 4}

# Checking membership
print(3 in numbers)  # True
print(5 in numbers)  # False
```

**Program output:**

```
{1, 2, 3, 4}
{1, 3, 4}
True
False
```

### 9.1.4

How many elements will remain in the set after performing the following operations? What number does the code print?

```
A = set()
A.add(100)
A.add('Peter')
A.add(1000.56)
A.remove("Peter")
A.add(50+50)
```

```
A.add("x")
print(len(A))
```

- 3
- 4
- 5
- error
- 2

### 9.1.5

#### Operations with sets

Sets are ideal for performing mathematical operations like union, intersection, and difference.

The **union** ( $\cup$ ) of two sets combines all the unique elements from both sets into a single set. No duplicates are allowed; repeated elements appear only once in the result.

- Group A: {apple, banana, cherry}
- Group B: {banana, cherry, date, fig}
- The union of these groups will include every unique item: {apple, banana, cherry, date, fig}.

The **intersection** ( $\cap$ ) of two sets contains only the elements that are common to both sets. If an item is not present in both sets, it will not be included in the intersection.

- Group A: {apple, banana, cherry}
- Group B: {banana, cherry, date, fig}
- The intersection of these groups includes items found in both groups: {banana, cherry}.

The **difference** ( $-$ ) of two sets contains elements that are in the first set but not in the second set. The result depends on the order of the sets in the operation.

- Group A: {apple, banana, cherry}
- Group B: {banana, cherry, date, fig}
- The difference of Group A - Group B includes items in Group A only: {apple}.
- Similarly, the difference of Group B - Group A includes items in Group B only: {date, fig}.

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
# Union: Combine all unique elements
print(A | B) # {1, 2, 3, 4, 5, 6}
# Intersection: Common elements
```



```
print(A & B) # {3, 4}
# Difference: Elements in A but not in B or opposite
print(A - B) # {1, 2}
print(B - A)
```

**Program output:**

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{5, 6}
```

### 9.1.6

What will be the result of the following code:

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m1 & m2
print(m3)
```

- {'C', 'A'}
- {'A', 'B', 'C', 'D'}
- {'A', 'Y', 'C'}
- {'B', 'D', 'C', 'X', 'Y', 'A', 'Z'}

### 9.1.7

What will be the result of the following code:

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m1 | m2
print(m3)
```

- {'B', 'D', 'C', 'Y', 'A'}
- {'A', 'B', 'C', 'D', 'C', 'Y', 'A'}
- {'A', 'Y', 'C'}
- {'B', 'D', 'C', 'X', 'Y', 'A', 'A'}

### 9.1.8

What will be the result of the following code:

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m1 - m2
print(m3)
```

- {'B', 'D'}
- {'A', 'Y', 'C'}
- {'A', 'B', 'C', 'D'}
- {'Y'}

### 9.1.9

What will be the result of the following code:

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m2 - m1
print(m3)
```

- {'B', 'D'}
- {'A', 'Y', 'C'}
- {'A', 'B', 'C', 'D'}
- {'Y'}

## 9.2 Comparison

### 9.2.1

#### Set comparisons

Sets can be compared using specific rules that help determine their relationship.

- Equal sets - contain exactly the same elements.
- Subset/superset - smaller or larger sets based on element inclusion.
- No Relationship - if sets have exclusive elements, no comparison is valid.

These comparisons are useful in programming when analyzing groups of unique items.

#### Equality of sets

- Two sets are equal if they contain exactly the same elements.
- $\{1, 2, 3\} == \{3, 2, 1\}$  - order does not matter in sets

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = {'A', 'C', 'B', 'D'}
print(m1 == m2)
print(m1 == m3)
```

**Program output:**

```
False
True
```

 9.2.2

Which pairs of sets are equal

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'B', 'C'}
m3 = {'A', 'C', 'C', 'D'}
m4 = {'B', 'A', 'C'}
```

- m2, m4
- m1, m2
- m1, m3
- m1, m4
- m1, m3
- m3, m4

 9.2.3**Subset relationships**

- A set is smaller than another set (a subset) if all its elements are also present in the other set.
- $\{1, 2\} \subseteq \{1, 2, 3\}$  -  $\{1, 2\}$  is a subset of  $\{1, 2, 3\}$

**Strict superset relationships**

- A set is larger than another set (a strict superset) if it contains all the elements of the smaller set plus additional elements.
- $\{1, 2, 3\} \supseteq \{1, 2\}$  -  $\{1, 2, 3\}$  is a strict superset of  $\{1, 2\}$

**No comparison possible**

- If one set has elements not present in the other, and vice versa, then neither is larger nor smaller.
- $\{1, 4\}$  and  $\{2, 3\}$  - no subset or superset relationship exists.

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'B', 'C'}
print(m1 > m2)
print(m1 < m2)
```

**Program output:**

```
True
False
```

```
m1 = {'A', 'B', 'C', 'D'}
m3 = {'A', 'B', 'Y'}
print(m1 > m3)
print(m1 < m3)
print(m1 == m3)
```

**Program output:**

```
False
False
False
```

### 9.2.4

Choose the true statement

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
```

- neither statement is valid
- $m1 < m2$
- $m1 > m2$
- $m1 == m2$

### 9.2.5

Choose the true statement

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'D', 'C'}
```

- neither statement is valid
- $m1 < m2$
- $m1 > m2$
- $m1 == m2$

## 9.3 Working with set

### 9.3.1

Sets are unordered, so the elements might not be visited in the order they were added. This is different from lists, which maintain the order of insertion.

If the set is for example {1, 3, 2}, the iteration order might be 1, 2, 3 or another variation.

Sets in Python are iterable, meaning we can loop through their elements one by one, just like with lists or tuples. This allows us to perform actions on each item in the set, such as printing, modifying, or performing calculations.

We can use a for loop to visit each element in a set.

```
# Looping through a set
colors = {"red", "blue", "green"}
for color in colors:
    print(color)
```

**Program output:**

```
blue
green
red
```

### 9.3.2

Complete the code that prints the list of words whose number of characters is greater than 3

```
my_set = {'Mom', 'had', 'Ema', 'at', 'home', 'for', 'one',
'hour'}
for _____:
    if _____(i) _____ 3:
        print(i)
```

- my\_set
- len
- size
- i
- in
- of
- >
- ==
- length
- <|

### 9.3.3

Sets in Python are inherently **unordered**, meaning the sequence of elements is not guaranteed. If you need a consistent order or want to arrange the elements, you can convert the set into a list or tuple. This transformation allows you to manipulate the elements in a predictable sequence.

We can first convert the data and then work with them or order them:

```
colors = {"red", "blue", "green", "yellow", "magenta"}
```

```

for color in colors:
    print(color, end=" ")
print()

l_colors = list(colors)
for i in range(len(l_colors)):
    print(l_colors[i], end=" ")
print()

# Converting a list to a sorted list
sorted_colors = sorted(l_colors)
print(sorted_colors)

```

**Program output:**

```

red green yellow magenta blue
red green yellow magenta blue
['blue', 'green', 'magenta', 'red', 'yellow']

```

- or we can create ordered list directly from set:

```

colors = {"red", "blue", "green", "yellow", "magenta"}
for color in colors:
    print(color, end=" ")
print()

# Converting a list to a sorted list
sorted_colors = sorted(colors)
print(sorted_colors)

```

**Program output:**

```

red green yellow magenta blue
['blue', 'green', 'magenta', 'red', 'yellow']

```

By converting a set into a list or tuple, we gain control over the order of elements. Sorting further enhances usability, making it easier to work with data in a structured and predictable way.

### 9.3.4

Select correct ways to transform set **b** to list **a**:

```
b = {1, 2, 3, 4}
```

- a = sorted(b)
- a = list(b)
- a = [] + b
- a = [b]
- a += b

### 9.3.5

**Find out all the different digits that are in the given number.**

To determine all the different digits in a given number, we can use a set. Sets naturally eliminate duplicates, making them ideal for this task.

- we are not interested in how many times they are repeated, only which ones occur there
- so we will create a set to which we will add elements
- adding a duplicate element does nothing
- after passing the number, we will list ordered digits

```
# Input number
number = input("Enter a number: ")

# Create an empty set
ud = set()

# Add each digit to the set
for digit in number:
    if digit.isdigit(): # Check if the character is a digit
        ud.add(digit)

# Display the unique digits
print("Unique digits in the number:", sorted(ud))
```

**Program output:**

```
Enter a number: 258784444888Unique digits in the number: ['2',
'4', '5', '7', '8']
```

### 9.3.6

Complete the code that finds out which letters appear in the entered word, regardless of their size. The result will be displayed as ordered lowercase letters:

```
word = input()
vowels = set('aeiouAEIOU')
result = _____()
for char in word:
    if char _____ vowels:
        result.add(char._____())
print(_____ (result))
```

- sort
- upper
- lower

- sorted
- of
- list
- order
- in
- set

## 9.4 Programs (set)

### 9.4.1 List of participants

Write a program that reads from the input a list of last names of the tour participants. The number of participants is unknown, but ends with 0 at the input. It is possible that some participants are duplicated in the list.

Ensure that each participant is in the list only once, and after loading, list the list in alphabetical order.

```
input:
Pear
Small
Green
Pear
Cruel
0
output:
[Cruel, Green, Pear, Small]
```

### 9.4.2 A simple list

Write a program that reads from the input a list of numbers specified in the lines below. The list is terminated by the value 0.

Let the program load the data into a set-type list and then print it out. Create a second set in which you put only the odd numbers from the first set. Print the contents of the new ordered set again. Print both sets in ascending order.

```
input:
1
5
4
8
7
3
```



```
21
0
output:
{1, 3, 4, 5, 7, 8, 21}
{1, 3, 5, 7, 21}
```

### 9.4.3 Nice or Smart

The university organized the Nice and Smart competition. She created two groups of contestants, coding them with random numerical values for anonymity. Each competitor will receive a participation medal and at the same time will take part in a joint lunch.

At the entrance, in the first row there is a list of contestants registered for the handsome competition and in the second row a list of registered competitors for the smart competition.

Write a program that prints a list of all participants who will receive lunch (that is, each one only once) and in the next line a list of contestants who participated in both contests. Arrange the data in both cases (as numbers).

```
input: 1 8 -3 4
1 8 5
output: -3 1 4 5 8
1 8
```

```
input: 1 2 3 4
5
output: 1 2 3 4 5
empty set
```

### 9.4.4 Persons in the object (ID)

Write a program that will store a list of people who entered and left the building based on their personal numbers.

If a person who is already registered in the facility enters the facility, they should not be added to the list, if the person leaves the facility, they should be removed from the list.

If you try to remove a person from the list who is not on it, let it be written: "VIOLATION "+ number of the infringer. Use the set data type, with input coded as 1, output coded as 0.

Finally, print the list in ascending order via set listing.

Entry will be implemented as follows:

- the first value will represent the number of operations to be performed
- it will be followed by entry/exit operations and the person's personal number.

For example for input:

```
6
0 3
1 2
1 9
1 4
0 2
1 4
```

the output would be:

```
VIOLATION 3
{4, 9}
```

### 9.4.5 Persons in the building (surname)

Write a program that will keep a list of people who have entered and exited the facility based on their last names.

If a person who is already registered in the building enters the building, he should not be added to the list.

If the person leaves the facility, let him be removed from the list.

If you try to remove a person from the list who is not on it, let it be written: "VOILATION " + the person's name.

Use the set data type, with input coded as 1, output coded as 0.

Finally, print alphabetically via set printout.

Entry will be implemented as follows:

- the first value will be the number of operations to perform
- it will be followed by entry/exit operations and the person's personal number.

For example for entry:

```
6
0 Skalka
1 Hruska
1 Slivka
1 Kapusta
```

```
0 Hruska
1 Ceresna
```

the output would be:

```
VIOLATION Skalka
{Ceresna, Kapusta, Slivka}
```

### 9.4.6 Reward of the faithful

Two nightclubs have opened in the university town, which keep records of who visits them. Many visitors walk from one to the other, always looking for something, which complicates traffic and disturbs residents living nearby. The city has therefore decided to reward visitors who do not move at night. At the entrance, the names are separated by spaces, while the first line contains the visitors of the first company, the second the visitors of the second. Write a list of those who were faithful that day - they were only in one business.

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
```

### 9.4.7 Happy string

At the input there is a list of students who came for the exam. Professor has a good habit of having a text string generated every day, and students whose name starts with it get the exam automatically.

The second line contains the generated string.

Write a list of the names of students who remain in the exam and arrange it alphabetically.

```
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
```

### 9.4.8 Word generator

Write a program that, for a given list of characters, generates all words consisting of three **different CAPITAL** letters.

Print the result in alphabetical order.

Assume that the characters in the generator can be repeated and there can also be numbers in it. For example for entry:

```
AB/1C112A4
```

the will be result:

```
{ABC, ACB, BAC, BCA, CAB, CBA}
```

# Dictionary

## Chapter **10**

## 10.1 What is dictionary

### 10.1.1

A dictionary is a Python data structure that organizes data into key-value pairs. It works like a real dictionary where you look up a word (key) to find its meaning (value).

- Keys are unique identifiers used to access values. Keys must be immutable within the program, they can be strings, numbers or tuples.
- Values are pieces of data associated with each key. Values can be any data type, such as strings, numbers, lists, or even other dictionaries. We can change them during the existence of the item.

```
# Create a dictionary
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
}
```

```
print(student)
```

**Program output:**

```
{'name': 'Alice', 'age': 20, 'grade': 'A'}
```

We created a dictionary with name **student**, which stores information about a student:

- Key: "**name**", Value: "**Alice**"
- Key: "**age**", Value: **20**
- Key: "**grade**", Value: "**A**"

### 10.1.2

A dictionary is a data structure that organizes data into key-value pairs, where:

- keys are immutable, values are mutable
- keys are mutable, values are immutable
- both, keys and values are mutable
- both, keys and values are immutable

### 10.1.3

#### **Data access and modification**

The dictionary is dynamic, allowing us to update, add, or remove items as needed.

To retrieve a value, we can use the associated key. For example, if we have a dictionary with information about a student, we can use the key like "name" to access their name.

```
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
}

print(student["name"])
```

**Program output:**

Alice

To change the value, e.g. name we use assigning a new value to the given key:

```
student["name"] = 'Alicia'
print(student)
```

**Program output:**

```
{'name': 'Alicia', 'age': 20, 'grade': 'A'}
```

While assigning a value to an existing key changes the value part, assigning a value to a key that does not exist yet creates a new pair - a dictionary entry.

```
# Add a new key-value pair
student["subject"] = "Math"
student["grade"] = "A+"
print(student)
```

**Program output:**

```
{'name': 'Alicia', 'age': 20, 'grade': 'A+', 'subject': 'Math'}
```

It is also possible to delete an existing pair:

```
del student["age"]
print(student)
```

**Program output:**

```
{'name': 'Alicia', 'grade': 'A+', 'subject': 'Math'}
```

## 10.1.4

How can you add a new key-value pair to a dictionary?

- dictionary["key"] = "value"
- dictionary.add("key", "value")
- dictionary.insert("key", "value")

- `dictionary.append("key", "value")`

### 10.1.5

We can also use dictionary as a list with named values. Let's create e.g. list of employees with their salaries.

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
print(d)
```

**Program output:**

```
{'Adam': 1500, 'Beta': 1800, 'Cynthia': 1578, 'Damian': 1384}
```

The aim of the dictionary in this case is: creating a list of unique objects and assigning a value to them and subsequently, getting a pair for the unique value that corresponds to it For the unique Adam, we get his salary:

```
print(d['Adam'])
```

**Program output:**

```
1500
```

### 10.1.6

How many elements will be in the dictionary after performing the following operations

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
d['Beta'] = 2000
del(d['Cynthia'])
d['Milo'] = 1800
d['Bela'] = 1650
d['Cynthia'] = 1700
del(d['Milo'])
d['Charles'] = 1980
print(len(d))
```

- 6
- 4
- 5
- 7

### 10.1.7

#### Basic operations

A simple operation is to find the number of elements - we use the standard function `len()`:



```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
}
print(len(d))
print(len(student))
```

**Program output:**

```
4
3
```

Command **clear()** is used to clear the entire list.

```
d.clear()
print(d)
```

**Program output:**

```
{}
```

Note that the notation for an empty dictionary is the same as for a set - `{}`.

The difference between an empty dictionary and an empty set is how they are created:

- Empty dictionary: `{}` - by default curly braces without any elements define an empty dictionary in Python.
- Empty set: `set()` - you must explicitly use the `set()` function to create an empty set, because `{}` is reserved for dictionaries

Curly braces `{}` are used for both dictionaries and sets, but the context determines their purpose:

```
empty_dict = {} # This creates an empty dictionary.
empty_set = set() # This creates an empty set.
```

### 10.1.8

How do we distinguish between the creation of an empty dictionary and a set:

- `empty_dict = {}, empty_set = set()`
- `empty_dict = dict(), empty_set = {}`
- `empty_dict = {}, empty_set = {}`
- `empty_dict = dict(), empty_set = set()`

 10.1.9**Various data acquisition options**

A dictionary is created by pairing keys with values. We can access the value corresponding to a key by placing the key inside square brackets:

```
d = {"name": "Alice", "age": 25}
print(d["name"])
```

**Program output:**

```
Alice
```

But, if we try to access a key that doesn't exist, we'll encounter an error:

```
print(d["surname"])
```

**Program output:**

```
KeyError
'surname'
```

To avoid this, you can use safer alternatives.

The **get()** method provides a safer way to access values in a dictionary.

- If the key exists, it will return the associated value.
- If the key doesn't exist, it will **not raise an error** but instead will return a default value. By default, this value is **None**, but you can specify a different value as the second parameter.

```
print(d.get("name")) # Output: Alice
print(d.get("address")) # Output: None
```

```
# Using get() with a default value
```

```
print(d.get("address", "Unknown")) # Output: Unknown
```

**Program output:**

```
Alice
```

```
None
```

```
Unknown
```

Another method to access and remove a dictionary entry is **pop()**. The **pop()** method removes the key-value pair from the dictionary and returns the corresponding value. If the key doesn't exist, it raises a **KeyError**, unless you specify a default value as the second argument.

```
# Using pop() to remove and access an item
```

```
name = d.pop("name")
```

```
print(name) # Output: Alice
```

```
print(d) # Output: {'age': 25}
```

```
# Using pop() with a default value
address = d.pop("address", "Unknown")
print(address) # Output: Unknown
print(d) # Output: {'age': 25} (No "address" key)
```

**Program output:**

```
Alice
{'age': 25}
Unknown
{'age': 25}
```

These methods allow us to handle situations where the key may not exist in a safer and more controlled way.

### 10.1.10

What is the main advantage of using the **get()** method when accessing a dictionary value?

- It returns None when the key does not exist (or a default value if specified).
- It raises a KeyError when the key does not exist.
- It automatically adds the key-value pair to the dictionary if the key does not exist.
- It removes the key-value pair from the dictionary.

### 10.1.11

Which of the following are true about the **pop()** method in a dictionary?

- It removes the key-value pair from the dictionary and returns the value.
- It raises a KeyError if the key is missing and no default value is provided.
- It returns None if the key is not found.
- It adds a default value to the dictionary if the key is missing.

## 10.2 Dictionary iterating

### 10.2.1

#### Typical examples of use

Dictionaries are ideal for storing data where each key is unique and maps to a corresponding value. Some typical use cases include:

- **Identification number** → **Name of the person**: each unique ID number maps to a specific person's name.

- **Name of the person** → **Telephone number**: we can quickly look up a person's phone number using their name as the key.
- **Municipality** → **District**: a dictionary can map each municipality to its district for easy lookup.
- **Telephone number** → **Name of the person**: quickly retrieve the name of a person by their telephone number.
- **Town** → **GPS coordinates of the center**: town names can map to the GPS coordinates of their central location.
- **Domain address** → **IP address**: map a domain name (like "example.com") to its corresponding IP address.
- **VRN (Vehicle Registration Number)** → **Customer**: a dictionary can be used to link a vehicle registration number to the customer's details.

### 10.2.2

In the following example:

#### **Town, County**

where both town and country are simple variables, what is the key and what is the value?

- Town is key, Country is value
- Country is key, Town is value
- Town is key, Country is key
- Town is value, Country is value

### 10.2.3

A **dictionary** in Python is a collection of key-value pairs, where each key is unique and is associated with a value. When working with dictionaries, there are often situations where we need to access or manipulate the data. One common operation is **looping through a dictionary**.

When we loop through a dictionary, we are typically interested in either the **keys**, the **values**, or both the **keys and values**. Python provides simple ways to access each of these components while iterating.

#### **Iterating through keys**

By default, when you loop through a dictionary, you loop through the **keys**. This means that for every cycle of the loop, the current element is the key of the dictionary. If we need to access the corresponding value, we can use the key to retrieve it.

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
for i in d.keys():
    print(i, d[i])
```

**Program output:**

```
Adam 1500
Beta 1800
Cynthia 1578
Damian 1384
```

We can list the keys as an independent list or transform them into a field and use them separately:

```
print(d.keys())
my_list = list(d.keys())
print(my_list)
```

**Program output:**

```
dict_keys(['Adam', 'Beta', 'Cynthia', 'Damian'])
['Adam', 'Beta', 'Cynthia', 'Damian']
```

 10.2.4

Which of the following statements is true when looping through a dictionary?

- By default, you loop through the keys of the dictionary.
- You can only loop through the values of the dictionary.
- You must use the `.values()` method to loop through the keys.
- You can loop through keys and values at the same time, but only by using `.keys()`.

 10.2.5**Iterating through values**

If we are interested in only the **values** (and not the keys), you can use the `.values()` method. This will give you an iterator that yields only the values associated with the keys in the dictionary.

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
for i in d.values():
    print(i)
```

**Program output:**

```
1500
1800
1578
1384
```

As in the case of keys, we can also transform values into a field and then use them independently:

```
print(d.values())
my_list = list(d.values())
print(my_list)
```

**Program output:**

```
dict_values([1500, 1800, 1578, 1384])
[1500, 1800, 1578, 1384]
```

## 10.2.6

What does the `.values()` method return when used with a dictionary?

- A list of all the values in the dictionary
- The keys of the dictionary
- A list of all the dictionary items (key-value pairs)
- A set of all the values in the dictionary

## 10.2.7

### Looping through both keys and values

This is the most common approach when we need to perform actions that involve both the key and the value. For instance, when creating a report from a dictionary, where we display both the item names (keys) and their corresponding quantities (values).

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
for i in d.items():
    print(i)
```

**Program output:**

```
('Adam', 1500)
('Beta', 1800)
('Cynthia', 1578)
('Damian', 1384)
```

Looping through dictionaries allows for flexible, dynamic processing of data stored in key-value pairs. Since dictionaries are unordered collections, the order of elements isn't guaranteed, but this doesn't affect the ability to loop through them and access data efficiently.

This operation is commonly used when we need:

- to **search** for specific keys or values.
- to **modify** values based on certain conditions.
- to **filter** the data in the dictionary.

 10.2.8

What does the `.items()` method return when used with a dictionary?

- A list of tuples, each containing a key and its corresponding value
- A list of all the keys in the dictionary
- A list of all the values in the dictionary
- A set of tuples, each containing a key and its corresponding value

## 10.3 Typical examples

 10.3.1

### Searching information I.

Searching in a dictionary is one of the most useful operations in programming. The idea behind searching a dictionary is to find the correct pair of key and value that meets our needs.

Consider a dictionary where each key is a word in **English**, and its corresponding value is the **foreign language equivalent**. This is an example where the **key** is the word in one language (e.g., English) and the **value** is the translation of that word in another language.

Let's consider an example where we have a dictionary mapping English words to their Spanish equivalents.

```
# English to Spanish dictionary
translations = {
    "apple": "manzana",
    "book": "libro",
    "house": "casa",
    "dog": "perro"
}
```

If we want to find the Spanish word for **apple**, we simply access the dictionary with the key "apple":

```
en = "apple"
sp = translations[en]
print(f"The Spanish word for '{en}' is '{sp}'.")
```

**Program output:**

```
The Spanish word for 'apple' is 'manzana'.
```

In this case, the key is "apple", and the value is "manzana".

If we want to find which English word corresponds to the Spanish word "libro", we need to search through all the dictionary values. Here's an approach to achieve that:

```
sp_word = "libro"
for en, sp in translations.items():
    if sp == sp_word:
        print(f"The English word for '{sp_word}' is '{en}'.")
        break
```

**Program output:**

```
The English word for 'libro' is 'book'.
```

- **translation.items()** represents a list of pairs (tuples). So we can iterate through this list using two variables - the first represents the key, the second the value of the values of the individual elements.

### 10.3.2

What happens if you try to access a non-existent key in a dictionary using square brackets?

- It raises a `KeyError`.
- It returns `None`.
- It adds the key with a default value.
- It ignores the operation.

### 10.3.3

#### Handling missing keys or values

When searching for a key or value in a dictionary, it's important to handle situations where the desired element is not found. Here's how to manage situation with missing key:

```
# English to Spanish dictionary
translations = {
    "apple": "manzana",
    "book": "libro",
    "house": "casa",
    "dog": "perro"
}
en = "juice"
sp = translations.get(en, 'not found')
print(f"The Spanish word for '{en}' is '{sp}'.")
print("The Spanish word for 'home' is
",translations.get('home'),".")
```



```
print("The Spanish word for 'apple' is
",translations.get('apple'),".")
```

**Program output:**

```
The Spanish word for 'juice' is 'not found'.
The Spanish word for 'home' is None .
The Spanish word for 'apple' is manzana .
```

We will use the **get()** function, which, in its basic form, returns **None** if the specified key does not exist. Alternatively, we can provide a second parameter, which will be returned as the default value if the key is not found.

To find if a **specific value exists**, you can use the **in** operator with `dictionary.values()`:

```
sp_word = "libero"
if sp_word in translations.values():
    for en, sp in translations.items():
        if sp == sp_word:
            print(f"The English word for '{sp_word}' is '{en}'.")
            break
else:
    print("Value not found.")
```

**Program output:**

```
Value not found.
```

 10.3.4

What happens when you use the **get()** function with a non-existent key and no default value is provided?

- It returns None.
- It raises a `KeyError`.
- It removes the key from the dictionary.
- It creates a new key with a None value.

 10.3.5

Which of the following are valid ways to avoid an error when searching for a key that might not exist?

- Use the `get` method.
- Provide a default value with the `get` method.
- Use the `pop` method without arguments.
- Directly access the key using square brackets.

## 10.3.6

### Searching information II.

A dictionary can be part of a list and describe the properties of the objects in the list.

### Record information about students, find out how many men and women are on the list.

We will record about students: name, gender, year of birth, residence

- the data will not be placed in the unnamed columns of the table
- each element of the list will have a data description (key) and its value
- entering the key value is also possible in the following way:

```
s1 = {'name': 'Jozef', 'gender': 'man', 'year': 1998,
      'residence': 'Paris'}
s2 = {'name': 'Klement', 'gender': 'man', 'year': 1968,
      'residence': 'Prague'}
s3 = {'name': 'Jana', 'gender': 'woman', 'year': 1999,
      'residence': 'Bratislava'}
s4 = {'name': 'Juan', 'gender': 'man', 'year': 1988,
      'residence': 'Madrid'}
z = list()
z = z + [s1]
z = z + [s2]
z = z + [s3]
z = z + [s4]
print(z)
```

#### Program output:

```
[{'name': 'Jozef', 'gender': 'man', 'year': 1998, 'residence':
'Paris'}, {'name': 'Klement', 'gender': 'man', 'year': 1968,
'residence': 'Prague'}, {'name': 'Jana', 'gender': 'woman',
'year': 1999, 'residence': 'Bratislava'}, {'name': 'Juan',
'gender': 'man', 'year': 1988, 'residence': 'Madrid'}]
```

How to count men and women? The easiest way is iterate list and check gender for every item:

```
man_count = 0
woman_count = 0

# Iterate through the list and check gender
for person in z:
    if person['gender'] == 'man':
```

```

    man_count += 1
    elif person['gender'] == 'woman':
        woman_count += 1

print(f"Number of men: {man_count}")
print(f"Number of women: {woman_count}")

```

**Program output:**

```

Number of men: 3
Number of women: 1

```

 10.3.7

Complete the code to find the number of registered persons born before the specified year:

```

s1 = {'name': 'Jozef', 'gender': 'man', 'year': 1998,
      'residence': 'Paris'}
s2 = {'name': 'Klement', 'gender': 'man', 'year': 1968,
      'residence': 'Prague'}
s3 = {'name': 'Jana', 'gender': 'woman', 'year': 1999,
      'residence': 'Bratislava'}
s4 = {'name': 'Juan', 'gender': 'man', 'year': 1988,
      'residence': 'Madrid'}
my_list = _____ s1, _____, s3, s4 _____
border_year = 1980
cnt = 0
for person in _____:
    if person['_____'] _____ border_year:
        cnt _____ 1
print(cnt)

```

- ==
- =
- +
- [
- s1
- >
- s2
- {
- <
- year
- -
- my\_list
- ]
- }

 10.3.8

Complete the code to **increase** age of all registered persons:

```
my_list = [{'name': 'Jozef', 'age': 19 _____,
_____ 'name': 'Klement', 'age': 20}, {'name': 'Jana', 'age':
20 _____, {'name': 'Juan', 'age': 30}]
print(my_list)
for _____ in _____:
    person['_____'] = person['_____'] _____ 1
print(my_list)
```

- name
- name
- age
- -
- year
- }
- (
- age
- }
- }
- person
- my\_list
- year
- +
- \*
- {
- )
- )
- (

 10.3.9**Filtering data**

Imagine we have a dictionary of students with their names as keys and their grades as values. We want to filter out only the students who scored above 70.

```
students = { "Alice": 85, "Bob": 65, "Charlie": 72, "Diana":
90, "Eve": 68 }
filtered_students = {name: grade for name, grade in
students.items() if grade > 70}
print(filtered_students)
```

**Program output:**

```
{'Alice': 85, 'Charlie': 72, 'Diana': 90}
```

The students dictionary contains names and grades. A **dictionary comprehension** is a concise way to create a dictionary based on an existing one while applying a condition or transformation.

- `{}` - curly braces indicate that we are creating a dictionary.
- **name: grade** is the format for a key-value pair in the resulting dictionary.
- **for name, grade in students.items()** is the iteration over the students dictionary, where **students.items()** gives pairs of keys (name) and values (grade) and each key-value pair is unpacked into the variables name and grade.

### 10.3.10

Imagine we have a dictionary where the **keys** are product names and the **values** are their prices. We want to create a new dictionary with only the products that cost more than \$50. Fill in the following code:

```
products = {"Laptop": 1200, "Mouse": 25, "Keyboard": 45,
            "Monitor": 300, "Headphones": 70, "USB Cable": 15}
expensive_products = {_____ _____ for product, price
                      _____ products._____() if _____ > 50}
print(expensive_products)
```

- for
- items
- in
- product
- price
- :
- of
- ,
- keys
- product
- price
- values
- ;

## 10.4 Programs (dictionary)

### 10.4.1 List of residents

Write a program that will register the list of house owners in the settlement. Record them in the house number (as a number) and name (string) structure. The house number and the name are entered in the lines below each other.

Make sure the loading continues until the program reads 0 on the input.

If there is a duplicate in the list, update the name to the one entered later. Finally, print the list.

```
input:
9
Smith
10
Slivka
8
Sekerka
2
Smith
6
Bulla
10
Slivka
0

output:
{2: 'Smith', 6: 'Bulla', 8: 'Sekerka', 9: 'Smith', 10:
'Slivka'}
```

## 📖 10.4.2 Dictionary

Write a program providing the functions of a dictionary, which loads pairs of words and their translation. Pairs are printed in rows below each other. Make sure the loading continues until the program reads 0 on the input.

Then load the words from the input and print their translation until you hit 0 again. If you find the given word, write the translation, otherwise print "x"

```
input:
matka
mother
otec
father
syn
son
0
syn
dcéra
matka
0
```

```
output:
son
x
mother
```

### 10.4.3 Capital cities I.

Create an application that reads state - capital city pairs and prints the capital city based on the state input. At the input there will be a list of state and capital pairs - each on a separate line. The list will be terminated with a value of 0.

Subsequently, states will be entered at the input, for which the capital city will be printed. If the entered state does not exist in the list, x will be written. The list of queries ends with the value 0 again.

Finally, print the list in a structure organized by states.

```
input:
Hungary
Budapest
Slovakia
Bratislava
Slovakia
Bratislava
Great Britain
London
Spain
Madrid
Austria
Vienna
Austria
Vienna
Italia
Roma
0
Czechia
Great Britain
Slovakia
0

output:
x
London
Bratislava
```

```
{Austria=Vienna, Great Britain=London, Hungary=Budapest,
Italia=Roma, Slovakia=Bratislava, Spain=Madrid}
```

### 📄 10.4.4 Capital cities II.

Create an application that reads state - capital city pairs and prints the capital city based on the state input. At the input there will be a list of state and capital pairs - each on a separate line. The list will be terminated with a value of 0.

Print in a structure organized by states. Create an inverted structure from this structure so that when a city is entered, it prints the state it belongs to.

Subsequently, cities will be entered at the input, for which the state will be printed. If the specified city does not exist in the list, x will be displayed. The list of queries ends with the value 0 again.

Finally, print the list in a structure organized by cities.

```
input:
Austria
Vienna
Spain
Madrid
Hungary
Budapest
Austria
Vienna
Hungary
Budapest
Poland
Warszava
0
Warszava
Paris
Vienna
Warszava
Vienna
Budapest
0

output:
{Austria=Vienna, Hungary=Budapest, Poland=Warszava,
Spain=Madrid}
Poland
x
Austria
```



```
Poland
Austria
Hungary
{Budapest=Hungary, Madrid=Spain, Vienna=Austria,
Warszawa=Poland}
```

### 10.4.5 Car rental

Design a structure that will ensure the tracking of borrowed cars in the car rental office. The car will be identifiable by VIN and the person by name.

- If the car is rented by a person who already owns a car, it is fine, one person can rent more than one car.
- If someone wants to borrow the car that is rented, it is not allowed - it will say "OCCUPIED".
- Make it possible to return the car - then the car-person pair will be removed from the list.

Borrowing will be coded as 1, return as 0, the question whether the car is rented will be coded as 2 and the answer to it will be "YES" or "NO".

Entry will be implemented as follows:

- the first value will represent the number of operations to be performed
- it will be followed by input/output/query and VIN operations.

For example for the input:

```
6
1 NR12234 Skalka
1 NR12345 Hruska
2 NR11111
1 NR12234 Slivka
1 NR00000 Paradajka
2 NR00000
```

the output will be:

```
NO
OCCUPIED
YES
```

### 10.4.6 Dictionary manipulation

Design a structure that will ensure the storage of words in the English-Slovak dictionary. Make sure to add en/sk pairs.

The addition will be coded as 1, the query for an English word as 2, while the answer will be either the Slovak translation or x, if the word is not in the dictionary.

Entry will be implemented as follows:

- the first value will represent the operation to be performed
- it will be followed by space-separated input/output/query data
- query will be terminated with a value of 0

For example for the input:

```
1 mother matka
1 dog pes
2 cat
1 hen sliepka
1 son syn
2 dog
0
```

the output will be:

```
x
pes
```

### 10.4.7 Character sort

Write a program that will compute the number of occurrence characters in a given string. The input contains the string of characters.

Print each character of the string and the number of its occurrences (separated by a space) in the descending order of these numbers of occurrences.

```
input : Java
output: a:2 J:1 v:1
```

```
input : The string for character counting
output:  :4 r:4 n:3 c:3 t:3 h:2 i:2 a:2 o:2 e:2 g:2 f:1 s:1
T:1 u:1
```

### 10.4.8 Chemical Calculator

Write a program to calculate the mass of molecules of chemical compounds given by the formula.

At the input is the formula of the compound. It is true that each element in the formula can be uniquely identified - the mark has one or two characters and is

followed either by another mark or a numerical value expressing the number of atoms.

Data on the weights of chemical elements is contained in the text file `elements.csv` (we will be interested in the chemical symbol of the element, which is the second data and the molar mass of its atom, which is the last data in the line). Load the appropriate part of them into the dictionary.

```
elements.csv:  
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  
...
```

To calculate the mass of sulfuric acid ( $\text{H}_2\text{SO}_4$ ), we must add 2x the mass of a hydrogen atom (H), 1x the mass of a sulfur atom (S) and 4x the mass of an oxygen atom (O).

If an element tag that does not exist (not in our table) appears in the parsed formula, print an error message.

```
input: H2O  
output: 18.02
```

```
input: CO2  
output: 44.01
```

```
input: Hi3  
output: error
```

```
input: H2SO4  
output: 98.08
```



# PRISCILLA



[priscilla.fitped.eu](http://priscilla.fitped.eu)