FITPED | AI

Python - Interactive

Ján Skalka
Ľubomír Benko
Nataša Skalková
Eugenia Smyrnova-Trybulska

www.fitped.eu

2024

Funded by the European Union

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

FITPED

Python - Interactive

Published on

November 2024

Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Nataša Skalková | Teacher.sk, Slovakia

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

Reviewers

Piet Kommers | Helix5, Netherland

Oldřich Trenz | Mendel University in Brno, Czech Republic

Vladimiras Dolgopolovas | 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.

ISBN 978-80-558-2221-1

TABLE OF CONTENTS

Sandbox

Chapter 1

1.1 Sandbox

1.1.1

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.

You can write and switch on any code in the Python language. The icons in the upper part serve for pausing the software or in worse case for resetting the kernel, on which the software is running.

```
# write you code here
```

Introduction to Programming

Chapter 2

2.1 Problem and algorithm

2.1.1

Algorithm and problem

- the assumption for the use of algorithm is the existence of a problem
- in the first years of life, problem solving is built on different basis than later in life
- solving many problems is automatic, without being aware of the existence of the algorithm

2.1.2

The problem of drinking cocoa

- check whether there is enough ingrediencies
- pour milk into the cup
- heat it up
- mix sugar and cocoa in an empty cup
- check the temperature of milk
- if the milk is not warm enough, go back to the point 4
- pour the milk into the mixture of sugar and cocoa
- let it cool out
- drink it

And the problem is solved...

2.1.3

Terminology

- **Problem** - the state, in which there is a difference between what we have at the given moment and what we want to achieve
- **Problem solving** - removing the difference between the original state and the one we want to achieve
- **Algorithm** - procedure, we follow while solving the problem

It is not possible to solve every problem and we do not always get to the required outcome.

2.1.4

Algorithm and life

- solving real-life problems using algorithm principles is quite difficult
- correct algorithm always takes into consideration all the situations, details, accidents or uncommon situations (drinking cocoa, autonomous vehicle)
- talking about algorithms only makes sense when we have access to a limited set of rules (it can even be really huge), by which we can design the procedure of solution

2.1.5

Algorithm and programming

Algorithm - an elementary concept, a correct definition is necessary:

- an exact sequence of steps and instructions that will lead us from the (changeable) input data to the result in the final time
- not every sequence is algorithm, it has to meet following traits:

1. elementarity: the procedure is composed of simple steps that are understandable to the executor (computer, non-thinking device, human)
2. determinism: the procedure is structured in such a way that it is clearly determined what action should follow, or whether the procedure has already ended
3. efficiency: the calculation gives a result after a finite number of steps at every moment of its execution
4. finality: the procedure always ends after a finite number of steps have been performed
5. mass: the procedure is applicable to the entire class of admissible input data (can be applied to different inputs)
6. efficiency: the procedure is carried out in the shortest possible time and with the use of the least amount of resources (time and memory)

2.1.6

Elementality

Every procedure can be written in multiple ways:

It is important for the executor to understand the procedure.

Heating up the milk in a microwave oven:

- longterm owner: heat up the milk
- new owner?

- a child(wait for your parents and ask them)

Find the sixth square for number 2

- this formulation is OK for pupil of fifth or sixth year
- pupil in the second year would not be able to solve it, even though it is only 2.2.2.2.2.2

Exactness:

- Mill the two days old rolls!
- Crack two eggs!

2.1.7

Determinism

- in every step it has to be clear, where the solving is supposed to be going, how to continue
- if for example the sequence of "wash yourself, clean after yourself, take your clothes off and go to sleep" would be executed in different order the outcome could be different than it was meant to be
- if the non-thinking device did the most time consuming task at first: it would go to sleep and when it waked up it would clean, take off its clothes and wash itself

2.1.8

Efficiency

- execution of the algorithm demonstrably leads to the correct result after a finite number of steps when solving any of the group of tasks for which the algorithm was created
- in real life it doesn't always work - cooking/baking
- digital technology, as long as there is no malfunction, has no problems with this requirement

2.1.9

Finality

- characteristic requires that the algorithm always terminate
- a person working with a problem based on experience can determine whether or not his procedure will give the right result (or whether it will end or not)
- a computer without experience cannot decide at such a level

Algorithm for computer:

- put the pot of food on the stove
- turn on the gas
- stir until it starts to boil

Or:

- dig until you find the treasure
- as long as the entered number is less than one, multiply it by two!

There are also problems, whose solution is finite, but finding the result takes a very long time.

- encryption algorithms, when theoretically we can decrypt every message, but the implementation time is so long that the message loses its meaning after decryption (after 10 years)
- counting the cells in human body
- molecules in liter of air
- grains of sand in the desert

2.1.10

Mass

- feature more useful than necessary
- not every algorithm can be mass
- it necessary to put some input parameters into the algorithm
- e.g. block volume, calculation of the length of the braking distance at the speed of the vehicle and the road surface and the specified weight
- when creating an algorithm, it is not about solving a specific problem, but only about describing a procedure that can be used to obtain a result

2.1.11

Efficiency

- again not necessary feature
- to design a procedure that can solve the problem by using the minimal amount of resources in the shortest possible time
- even an inefficient algorithm is an algorithm

During the parade, the commander needed to find out the number of soldiers on board. The soldiers stood in 32 rows of 17. He assigned the task to two soldiers.

- The first one proceeded as follow: 17+17+17+17...+17
- The second one tried 17x30

What do you think, who was the first one to get the result?

The problem of adding numbers from 1 to 100:

- The first way, one that everybody understands is to proseed 1+2+3+4+...+0
- we will eventually get the result, but if we take the pair of numbers 1+100, 2+99, 3+98... 50+51 (together there is 50 of them) we solve the problem significantly faster: there is 50 pairs, their addition is 101, therefore 101 x 50 = 5 050

2.2 Why algorithmize?

2.2.1

Why algorithmize?

Life is an algorithm (resp. it is composed of algorithms).

Me waking up:

- Alarm rings
- I open my right eye
- I hit it (the alarm)
- I close the right eye
- I wake up suddenly in twenty minutes
- I eat breakfeast in a rush
- I run to work

What is the description of an algorithm good for?

- thanks to the description we can delegate another person or computer with the execution of the algorithm
- thanks to the expression of thoughts, the problem becomes more comprehensible to us and we are able to understand it better

2.2.2

Algorithmization

- process we are performing while writing/preparing an algorithm

Firstly we always need to think about:

- input conditions (f.e. the number of values that can enter the algorithm)
- output conditions (features of outcomes)

2.2.3

Algorithm language

Language - way of communication

- slovak language contains approx. 110.000 words
- english language contains approx. 800.000 words

Why the use human language is problematic:

- frazeologisms, proverbs and sayings
- synonymums
- homonymums
- forms, cases, persons
- smooth and natural development, adding and excluding words

=> the need for reduction of natural language

2.2.4

Algorithmic languages

Several types:

- flowcharts (sequence of activities described through graphic signs and text in them, defined by standards)
- structure diagrams (a condensed version of flow diagrams, which is not defined by the standard)
- picture languages (often children's programming languages allowing programming by connecting pictures)
- decision tables (describe more complex problems, consist of a list of conditions, a combination of conditions, a list of activities and a combination of activities - they are not suitable for our work)
- verbal notation of the algorithm in the natural language (formalized languages that differ from programming languages by the use of national words)
- programming languages (formalized algorithmic languages often based on word reduction of the English language)
- modelling languages (e.g. UML describes procedures by combining text and agreed symbols)

2.2.5

Composition of an algorithmic language

Operating component:

- commands - sentences of the language commanding the processor to perform specified actions (input, output and assignment)
- variables and constants - objects containing values during the implementation of the algorithm
- expressions - description of constants, variables and values use; how to process them using operations and functions similar to mathematics e.g. content = a * b, content2 = pi * r * r

Control component (algorithmic structures):

- tools for managing the sequence of execution of individual elements of the operational component
- thanks to it, the action to be performed is clearly determined in each step of the algorithm sequence
- a sequence of commands
- branching - decision making based on a condition
- cycle/loop – repetition of activities

2.2.6

Commands notation

- notation using natural language is not clear
- that is why we use programming language...

2.3 Programming language

2.3.1

Programming language

- **algorithmic language** - allows writing the sequence of command execution
- **programming language** - formalizes the algorithmic language into a notation that can be processed by a translator; based on the notation, commands are executed by a non-thinking device (usually a computer)
- **programming** - analogy of algorithmization, but additionally with the notation in the relevant programming language

2.3.2

Types of programming languages

Low-level programming languages:

- closer to the machine language
- easier/faster execution of the instructions
- often tied to the processor's instruction set

- machine code, assembler, maybe C

Higher level programming languages:

- more user-friendly
- closer to the human language - more abstract communication
- translation to the instructions of the processor
- often multiplatform
- Java, C, Pascal, Python, JavaScript, PHP

2.3.3

History of the programming languages

- 1940s - machine language programming
- 1950s - programming in the language of symbolic adresses
- 1956 - programming language fortran (Formula TRANslation)
- 1958 - programming language algol (ALGOrithmic Language)
- 1961 - programming language basic (Beginners All-Purpose Symbolic Instruction Code)
- around 1970 - programming language pascal (systematic, structured programming)
- around 1980 - programming language C (+ transition to object-oriented programming)
- 1990s - development of the more complex versions of languages with the aim of using new possibilities, especially of personal computers (graphic, sound, multiamedia) - event-driven programming (Visual Basic, Delphi)
- present - programming languages independent on platform (Java, script languages - PHP, Python), .NET platform a frameworks (e.g. Javascript)

2.3.4

Program life cycle

- **problem definition** - a clear problem definition is half the solution
- **problem analysis** - dividing the problem into smaller and simpler subproblems, complete definition of inputs and outputs
- **solution design** - description of the sequence by steps in human/algorithmic language
- **coding** - programming - transcription of the design into source code
- **testing** - removing program errors (syntactic and logical)

deployment at the customer and provision of support

2.3.5

Program execution

The device executes commands based on their translation into the language of the device - most often **the machine code of the processor**.

Translation can be executed in two ways:

- **before first time run** - trough the compiler
- **in real-time** - trough the interpreter

2.3.6

Compiler

- the source code translates into machine instructions (exe file)
- we named translator the compilator, the process of translation is compilation
- file containing commands in the language of the executing device is created (machine code)
- errors in code have to be removed before the start of process
- examples: C, Pascal

2.3.7

Interpreter

- source code is translated into the instructions while the program is running
- interpreter translates it for the processor command after command
- a group of basic syntax errors is revealed at the beginning, some of them only while the program is running
- Basic, Java, PHP, Python

Python

Chapter 3

3.1 Python language

3.1.1

Python

- language creation: 1989
- introduced by Guid van Rossum
- high-level language - to express the commands it uses certain words from english language
- a translation of commands to a low-level, that the computer understand is executed only when the the program is started - code is interpreted
- designed so that programs can be created using a minimum of code

3.1.2

Program behavior

- Python is case sensitive - it distinguishes between the lower and upper case letters
- Interpreter considers following commands to be different:

```
Command
command
COMMAND
```

- interpreted code => starting the program does not mean the program is error free
- errors only appear at the moment when the interpreter comes to a line of code with error notation

- **WARNING**, it may happen that the program you create will work at first, but an error and error message will appear while the program is running
- e.g.

```
print("Hello")
print("I am the computer No." + 10)
```

3.1.3

First program

Write a program that says hello.

This task has nothing to do with algorithmization, it serves to:

- familiarization with the basic syntax of the Python language
- presentation of the procedure for creating the program
- familiarization with the method of outputting results from the program

3.1.4

- in every programming language we need to follow certain rules
- it is not enough to write commands one after the other
- in the Python language we write one command in one line

So let's try to find the solution for the given task - **Write the program that says hello.**

The **print()** command is used to print the text.

We enter the text that we want to print in brackets and enclose it in quotation marks.

```
print("hello")
```

By clicking the arrow icon behind the program you start the program and below the code you will see its result - try it.

You can experiment with the program and rewrite the text between quotation marks to different greeting.

After running the program again the result below the code will change.

The number before the arrow indicates the number of program starts - it is only informational, it does not limit the user in any way.

3.2 Output command

3.2.1

We continue to say hello

Create a new program, in which you say hello and subsequently introduce yourself.

The situation is very simple:

- we say hello in the first line
- in next line we repeat the command for printing and we enter "introduction" in quotation marks

```
print("Hello")
print("I am Jose Carrot")
```

- commands are separated by the newline character = we place each command on a new line

Try to add your hometown to a new line by using the print() command.

3.2.2

- each **print()** function always prints its text on a new line

```
print("Mother")
print("Father")
```

- to display the output with multiple values in one line you need to separate the values with a comma

```
print("Hi!", "I am a super student.")
```

At first glance, this entry does not differ in any way from:

```
print("Hi! I am a super student.")
```

it just inserts a space between the two texts

However, later it will prove to be very useful...

3.2.3

String

Text in quotation marks, or any sequence of characters or digits, which we will use as written text (not a number) is denoted as a **string** in programming language.

The easiest way to identify a string is that it is placed between delimiting characters, quotation marks("") or apostrophes(').

```
"string 1"
'string with apostrophes'
```

3.2.4

Python language only supports two ways of notation of strings - delimitation by quotation marks " or apostrophes '.

```
print("Hello world!")
```

```
print('Hello world!')
```

What is it good for?

If you decided to print the text which is supposed to contain quotation marks, for example:

```
I am a "programmer"
```

we would be facing a problem because the entry

```
print("I am a "programmer".")
```

would be invalid.

Python evaluates the quotes behind "I am" as the end of the string and analyze the rest as a new string looking for example for a comma as a separator.

The solution is to use apostrophes - the text for the output starts with an apostrophe and unless Python finds a closing apostrophe it prints content as entered.

```
print('I am a "programmer".')
```

The opposite version, where we use an apostrophe inside a string is also possible.

```
print("I am a 'programmer'.")
```

3.2.5

- the **print()** function provides versatile functionality
- without the content it prints an empty line or skips line
- e.g.:

```
print('next line will be empty')
print()
print('a line is omitted before this')
```

3.2.6

- special ability of the **print()** command is that it is able to evaluate (calculate) an expression
- for now we will limit ourselves to mathematical examples
- e.g.:

```
print(20 + 10)
```

Program output:

```
30
```

...at first it finds out the result of the calculation in brackets and then prints it

We enter the calculation **without the quotation marks**, based on which the system knows that it is supposed to work with the content of the brackets as numbers and we do not just want to print it in in the same form as we see it between the quotation marks.

For code:

```
print("20 + 10")
```

Program output:

```
20 + 10
```

the result is the same as the text in the quotation marks.

Variables

Chapter 4

4.1 Variable

4.1.1

- so far we have written informations directly in "program".

```
print("hello")
```

- or we performed calculations

```
print(956 + 3)
```

- we did not remember any informations so we could not work with them on several parts of the program

4.1.2

- if we want to remember the value for later or multiple use, we need to use so-called **variables**

Variables

- they represent an independent (named) place in computer's memory
- we can use one or many of variables in program
- each variable has to have its own unique name defined by the programmer

Variable is created by executing an **assigment command**

- based on the name placed on the left side
- the value entered on the right side is inserted (assigned) to it

It is not needed to define earlier if we use a number, characters or sequence of characters - string

Python takes care of correct setting of the variable automatically.

```
n = 10
n=10
n= 10
myname = 'George'
```

All lines assigning the number **10** to the variable **n** are accepted. Spaces are simply ignored.

There are rules for formal modification of source code, that prefer code where is one space on both sides of a command (PEP8, https://www.python.org/dev/peps/pep-0008/).

Variable **can not** exist without being assigned a value.

4.1.3

In variable names the compiler distinguishes between lower and upper case letters, variables:

```
myvariable
Myvariable
```

are two different variables.

4.1.4

Variable name

- can be practically any one

```
a
counter
Fero12
Bratislava_is_the_city
Be_careful_it_is_bad
number
_number
Number
Number_1
number_Second
```

It is necessary to follow rules:

First character has to be:

- **alphabet latter** (upper or lower case), letters from the alphabet of different languages can also be used (but it is not recommended)
- **underline character** "_"

Rest of the name can consist of letters, underscore character "_" and digits.

Unacceptable variable names are e.g.:

```
4pieces
#number
```

- it is recommnded to use lower case letters and underscore character for variable names

4.1.5

Variable names **cannot** be the same as **Python** keywords.

The list of the keywords is as follows:

True	False	class	def	return
if	elif	else	try	except
raise	finally	for	in	is
not	from	import	global	lambda
nonlocal	pass	while	break	continue
and	with	as	yield	del
or	assert	None		

4.1.6

Variable **is created** by executing an **assignment statement**

Assignment statement:

- inserts some value into the variable

It consists of:

- the variable name of the left side
- operator "="
- the value (or calculation, based on which the value is obtained) on the right side

We will initially insert a number or text into the variable.

```
n = 10
var_name = 'word'
```

4.1.7

- we can find out the content of the variable by using the **print()** command
- if we state its name in the brackets (without the quotation marks), it will print the content of the variable

```
print(variable)
```

Program output:

```

```

- we do not enclose the variable name into the quotes or apostrophes

- according to the use or not of the quotes characters and apostrophes the interpreter knows whether it is supposed to print the content of variable or the text which we wrapped in quotes or apostrophes

```
weight = 73
print(weight)   # prints 73 - value inserted in variable
weight
print("weight") # prints text weight
```

4.2 Expression

4.2.1

Expression

- we often store the result of the calculation in variables
- such a notation is referred to as an expression

E.g. in program:

```
x = 10 + 20 - 3 * 7
```

the is used common mathematical expression, which is evaluated from left to right, observing the priority of mathematical operation, where the product (*) takes priority over the sum (+) and the difference (-).

The result stored in variable x can be printed as:

```
print(x)
```

In an expression, like in mathematics, parentheses take priority during evaluation. For example in the command:

```
x = 10 + (20 - 3) * 7
print(x)
```

Program output:

```

```

the difference in parentheses is evaluated first, then the product and finally the sum, * takes precedence over + and -.

The difference between directly printing the result of the calculation using

Program output:

and storing it to the variable means that we only see the result when it is printed and if it is saved to a variable, we can use it later.

E.g.:

```
print(x*x)
print(x/2)
```

Program output:

4.2.2

In the expression on the right side we can also use variables.

E.g. in program:

```
x = 10
y = 20
z = x + y   # values are used instead of variable names: 10 + 20
print(z)
```

The sequence is as follow:

1. firstly we insert values into the **x** and **y** variables
2. the calculation is based on the expression **x + y** - instead of the variables, their values is used - we will perform the sum of the values stored in the variables **x** and **y**. i.e. 10 + 20
3. the result is inserted into the variable, whose name is placed on the left side, i.e. variable **z**
4. we print the content of variable **z**

Variable that are listed to the right side of assignment symbol, are always replaced by the value they contain during the calculation, and their contents are not changed by this use.

4.2.3

- if we use the name of the variable somewhere other than on the left side of the assignment expression, then the value that the variable contains is replaced it
- we can freely combine directly entered values and variables in expressions, e.g.:

```
mysum = 100
new_sum = mysum - 20
print(new_sum)
```

The value **80** will be stored in the **new_sum** variable after the commands are executed.

4.2.4

We can often encounter notation where the name of the same variable appears on both sides of the assignment command

```
poc = 3
print(poc)
poc = poc + 1
print(poc)
```

The calculation procedure is the same as in the previous cases. The right side of the assignment command is processed first:

- the current value of the **poc** variable is used
- the value **1** is added to it
- the result of the expression is then stored into the **poc** variable, overwriting its original value

So the value **3** is printed first, it changes, and the value **4** is printed in the second **print()**.

4.3 Output formatting

4.3.1

Just as we could execute the calculation when listing the values, we can also execute it with variables. We can **print** not only the values of the variables but also the result of the operation.

So instead of:

```
a = 15
b = 10
c = a + b
print(c)
```

we can skip the calculation of the variable **c** and directly print the sum of two variables.

```
a = 15
b = 10
print(a + b)
```

4.3.2

If we want to print the values of several variables, it is possible to print them in one command.

```
x = 10
y = 20
z = x + y
print(x, y, z)
```

prints out:

```
10 20 30
```

The space between the values is inserted automatically by the **print** command.

4.3.3

Program printout:

```
x = 10
y = 20
print(x, y, x + y)
```

has the form:

```
10 20 30
```

- is very brief
- the sequence of numbers in the printout may be unclear to the user
- that is why is advisable to combine variables printout with descriptive texts, that explain the numbers in more detail, e.g.:

```
x = 25
print('Value of variable x is', x)
```

It prints out:

```
Value of variable x is 25
```

- the variable **x** enclosed in apostrophes is an ordinary **character string**, therefore we do not assign any value to it in print command

- the value is only assigned to the stand-alone variable name specified as the second parameter of **print**

4.3.4

- combining variables with static texts requires separating each part of the printout with a comma, e.g.:

```
x = 10
y = 20
z = x + y
print(x, '+', y, '=', z)
```

- the result will be as expected:

```
10 + 20 = 30
```

- when constructing the printout, do not forget that there is a space between the individual parts of the printout
- in the printout it is added automatically thanks to the rules defined for the **print** command

Input Command

Chapter 5

5.1 Input

5.1.1

Write a program that calculates the content and circuit of a rectangle.

Calculating the area of the rectangle for the specific values is easy,

e.g.:

- for 4 and 3 we put the values into the variables
- we assign the calculation to the variable
- we print the result

```
a = 4
b = 3
content = a * b
print(content)
```

If we want the calculation for the values 5 and 7 do we need to write a new program?

By the way, to get the result in specific case it would be enough to leave only the multiplication itself on the program...

```
print(5 * 7)
```

5.1.2

- one of the "useful" properties of the algorithm is **mass**
- it allows us to use the same algorithm for solving all tasks of the same type
- for this approach we need to generalize the approach to the problem
- if it is mathematical problem, it has often been generalized by someone before us

```
S = a * b
```

- it means, that we calculate the area as a product of the dimensions of the sides
- the size of the first side represents designation **a**, the second **b**
- **a** and **b** are what we can think of as variables in programming language

5.1.3

- writing the value of the variables to the code is nonsens
- we need to teach the program to read input values from the user without changing the code

The **input** command is used to get the values of variables from the user

- the **input** command reads the data entered by keyboard and confirmed by *Enter* and **move** it to the program as a text string
- the command allow us to write and delete input value and only after the press *Enter* send the data to the program
- then we insert the data into a variable

The entire entry has the form:

```
data = input()
```

Before pausing the program and waiting for the input, it is usually necessary to inform the user what input is expected, e.g.:

```
print('Write the name: ')
first_name = input()
```

Next, we can work with variable, e.g.:

```
print('Hi, you are', first_name)
```

5.1.4

The instructions for the user also can be enetered in the parantheses of the **input** command.

```
firstname = input('Write name: ')
surname = input('Write surname: ')
```

We can print the variables loaded using the **input** command in the same way as before, using the **print** command.

```
print('Hi', firstname, surname)
```

Print the text according to the entered first name and last name, e.g.:

```
Hi Jose Carrrot
```

5.2 Numeric input

5.2.1

Let's try to execute the following command, whose goal is to sum up two numbers.

```
a = input('Write 1st number: ')
```

```
b = input('Write 2nd number: ')
print(a + b)
```

- for **3** and **2** we expect **5**
- the result is **32**

- the **input** command always returns the value as a **text string**
- it concat the contents of the variables as text = it merges them

5.2.2

- if we are sure that a number will be entered at the input, we need to **convert** the data
- conversion = data type change: text -> number, number -> text etc.
- conversion from text to number is provided by **int()**

The program will look like this:

```
text1 = input('Enter the 1st number: ')  # rads the TEXT
enterd at the input
a = int(text1)                           # changes the
originaly entered text to a number
text2 = input('Enter the 2nd number: ')  # reads the second
TEXT entered at the input
b = int(text2)                           # second text is also
changed to a number
print(a + b)                             # finds/calculates
the result for the numbers
```

- **int(text)** provides the trasformation of text into a number
- it inserts numerical value returned by the **int()** function into the variables **a**, **b**

With such values the sum operation performs mathematical addition.

If there is text stored in the variables, the operation "+" will merge them.

```
print(text1 + text2)
```

Attention, if one value is text and the other is numeric, the program will throw an error.

```
print(text1 + b)
```

Program output:

```
NameError
name 'text1' is not defined
```

5.2.3

Let's get back to the task:

Write a program that calculates the area and circuit of a rectangle.

```
t1 = input('Enter the side a: ')
a = int(t1)
t2 = input('Enter the side b: ')
b = int(t2)
S = a * b
print('Area: ', S)
```

5.2.4

Write a program, that prints 5 multiple of the entered value:

```
```

Conditional Statement

Chapter 6

6.1 Conditional statement

6.1.1

- **sequence -** a sequence of commands that is executed in the order in which it is entered in the program
- the compiler proceeds one by one and when the command is executed it moves to the next one

```
aa = input('Enter the first value')
bb = input('Enter the second value')
a = int(aa)
b = int(bb)
print('The sum is:", aa + bb)
```

- most programs need to **decide** how to proceed further based on the processed data

6.1.2

The option to decide and execute different commands based on the fulfillment or unfulfillment of the condition is provided by branching.

It consist of a condition and commands that are executed in case of fulfillment and non-fulfillment of the condition

The branching command has the following form:

```
if condition:
      command
```

The condition represents a notation that can be said to be true or false

It often takes the form of:

comparing two values

- comparing the value and the variable (value that variable contains)
- e.g.:

```
5 > 3
8 < 1
a > 3
```

In code can takes the form:

```
if age < 10:
```

```
   print("minor")
```

```
if age > 18:
    print("adult")
```

In code it can take the form:

the basis of branching is the **if statement**

- it is followed by a condition that must result in true or false
- the condition must be followed by a colon
- the colon is followed by commands to be executed if the condition is met
- command, or commands are only executed if the condition is met
- if the condition is not met the command is not executed and the program continues with the next command

The command or commands must be indented from the marging using spaces.

If there is more than one command, the number of spaces must be the same, e.g.:

```
if age > 18:
    print("adult")
    print("because they are older than 18")
```

6.1.3

Incomplete branching describe the situation in which the condition is met.

```
if age > 18:
    print("adult")
```

Complete branching describes the behavior not just in the case the condition is met, but also if it is not:

- we add **else** to the original form of the statement
- the statement has the form:

```
if condition:
    commands _for the fulfilled condition
else:
    commands _for the unfulfilled condition
```

If and else statements must have the same indentation - in this case they start on the left margin,

e.g.:

```
if age > 18:
    print("adult")
else:
    print("minor")
```

- if the value of the **age** variable is larger than 18, the text "adult" is printed, otherwise (the **age** is less than or equal to 18) the text "minor" is printed

6.1.4

- the part of the program, in case the condition is met = **positive branch**
- the part of the program in case the condition is not met = **negative branch**

```
if age > 18:
    print("adult") # positive branch
else:
    print("minor")  # negative branch
```

6.1.5

- the use of one command in positive branch or one in negative branch is not really common
- we usually need to use more than one command
- the fact that several commands are to be executed in a certain branch is provided by indentation - it defines the whole **block of commands**

```
if condition:
       command1
    command2
    command3
else:
    command4
    command5
another_code
```

- commands 1-3 are executed in case the condition is met
- commands 4-5 in case it is not

The next code must continue with the same indentation as the **if** and **else** statements and it will be executed whether the condition was met or not.

6.1.6

Comparison

We have several operators for determining the relationship between the right and left sides:

== compares whether the values are equal to each other, e.g. **a==b**

- **<=** compares whether the value on the left is less than or equal to the value on the right, e.g. **c<=10**
- **>=** compares whether the value on the left is greater than or equal to the value on the right, e.g. **c<=10**
- **!=** compares whether the values are not equal, e.g. **a != b** - the condition is met if the values are different from each other

In case of using **<=** and **<=** signs their order must be observed. Using **=<** will be evaluated as an error.

```
if 4 =< 5:
  print('it applies')
```

Program output:

```
SyntaxError
invalid syntax (, line 1)
```

6.1.7

Let's test the values of two variables and print whether they are equal or one of them is larger than the other. We actually need to test three options.

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

By using **if**:

```
a = 10
b = 5
if a == b:
    print(a, b, 'are equal')
if a > b:
    print(a, 'is larger than', b)
if a < b:
    print(b, 'is larger than', a)
```

6.1.8

Let's redesign the solution to make it more efficient.

```
a = 10
b = 15
if a == b:
    print(a, b, 'are equal')
else:
    if a > b:
        print(a, 'is larger than ', b)
    else:
        if a < b:
            print(b, 'is larger than ', a)
```

- the condition **a > b** is tested only if **a == b** does not hold
- the condition **a < b** is tested only if a **== b** and **a < b** do not hold

However, the notation can be abbreviated because we do not have to execute the last test - the validity follows automatically.

```
a = 10
b = 15
if a == b:
    print(a, b, 'are equal')
else:
    if a > b:
        print(a, 'is larger than ', b)
    else:
        print(b, 'is larger than ', a)
```

6.1.9

- we also have the option to abbreviate the entry from the previous example
- there is a version of the **if** statement in the **if - elif - else** structure
- if the compiler evaluates one of the conditions as fulfilled, the following **elif** branches are no longer evaluated and it continues execution after the **if** block

```
if condition1:
    block of commands
elif condition2:
    block of commands

elif ...
elif ...
```

```
elif ...

else:
    block of commands
```

- the number of **elif** conditions is unlimited
- the **else** branch does not have to be entered

In the optimal entry, our solution would look like this:

```
a = 10
b = 8
if a == b:
  print(a, b, 'are equal')
elif a > b:
  print(a, 'is larger than', b)
else:
  print(b, 'is larger than', a)
```

Program output:

```
10 je väčšie ako 8
```

6.1.10

What is printed after the program is executed?

```
x = 6
if (x == 5):
    print('Hi')
    print('Hello')
else:
    print('Cheers')
print('Bye')
```

6.1.11

What is printed after the program is executed?

```
points = 7
if points == 10:
  print('excellent')
elif points == 9:
  print('super')
elif points == 8:
  print('okay')
```

```
elif points == 7:
  print('can be')
elif points == 6:
  print('a little above the line')
else:
  print('something needs to be done about it')
  print('end')
```

Cycle

Chapter 7

7.1 What the loop is

7.1.1

- we often need to repeat part of the algorithm
- the notation enables repeating = **loop**

for each repetition is important **what** (the body of the loop) is supposed to be repeated

- **when, for which values**, are the commands in the body of the loop supposed to be repeated

The loop enables repeating parts of the program for specified **list of values** or **until** the condition is met, e.g.:

```
_for values:  _for repetitions 1,2,3,4,5
what:               lift a barbell

_for values:  _for the coins  10,20,50,10,10
what:               put inside a purse

until when:     _while you have something _in your account
what:           buy toys

until when:     until you are at the end of the text
what:            replace the word five _with a number 5
```

7.1.2

Print the text 'Python' 7 times below.

- we need to repeat the printout of the value for 7 times
-

```
print('Python')
print('Python')
print('Python')
print('Python')
print('Python')
print('Python')
print('Python')
```

We can get the same effect by using loop.

in the definition of repetition we need to specify the group of values for which the printout is supposed to be repeated:

```
for i in 1, 2, 3, 4, 5, 6, 7:
   print('Python')
```

When repeating the same activity it is not important what values we enter, only their number is important.

7.1.3

How many times is the following loop executed?

```
for i in 1, 3, 4, 6:
   print('Winter')
```

7.1.4

Print "Hello" 5 times below

the number of repetitions does not depend on the values listed in the group but on their number

- we can provide the task with the following program:

```
for i in 1, 2, 3, 4, 5:
   print("Hello")
```

- **i** gradually acquires 5 different values and prints the text "Hello" ince for each of them
- however, the value of variable **i** is not used anywhere

Loop also fulfills the same role:

```
for i in 1, 1, 1, 1, 1:
   print("Hello")
```

or any other notation with the group of five not necessarily different values.

7.1.5

Definition of the loop

```
for variable in sequence:
    command
```

Must apply:

- the sequence must be defined as a sequence of values that we can loop trough
- there must be a colon at the end of the first line of the loop
- commands to be executed must be indented form the margin

Sequence of values:

values are separated by a comma, while each value has its place in the sequence, e.g. 1, 2, 3, 4, 5, 6, 7

7.1.6

Write the values 1-6 below.

we need to **repeat** the printout of the changing **value** 6 times

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
```

- the sequence of values that the loop is supposed to print are given in its definition:

```
for i in 1, 2, 3, 4, 5, 6:
  print(i)
```

- variable **i** acquires the values of the first element of the sequence and the command is executed, where the acquired value of the variable **i** is printed
- then we return to the beginning of the loop, the variable **i** acquires the value of the second element of the sequence and the command is executed again
- this is repeated until all sequence values are used up

7.1.7

Write the program that prints 5 lines with the text "I know how to use the loop now." and in each one of them it desplays the serial number of the line.

```
1 I know how to use the loop now.
2 I know how to use the loop now.
3 I know how to use the loop now.
4 I know how to use the loop now.
5 I know how to use the loop now.
```

- we need to print the values 1 - 5, so we insert the sequence 1 - 5 into the loop definition
- we ensure that the variable **i** gradually acquires these values

```
for i in 1, 2, 3, 4, 5:
```

- repetition consist in printing the variable **i**
- it is enough to add unchanging text to the changing numbers

```
for i in 1, 2, 3, 4, 5:
    print(i, 'I know how to use the loop now.')
```

7.1.8

Write a program that prints multiples for 1-10 for a given integer value.

We firstly ask the user for a number whose multiples we want to display.

```
text = input('Enter an integer from 1 to 10: ')
a = int(text)
```

- we need to provide following form for the output (e.g. for input 5)

```
1 - 5
2 - 10
3 - 15
etc.
```

- in the first line there is 1-multiple, in the second a double, in the third tripple, etc.
- the loop contains the output of the value from the enumerated sequence and its a-multiple

```
for i in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10:
    print(i,'-',a * i)
```

7.1.9

Adjust the program with the multiplier so that the lines are separated from each other by a line:

```
1 * 5 = 5
-----------------
2 * 5 = 10
-----------------
etc.
```

Original solution:

```
for i in 1,2,3,4,5,6,7,8,9,10:
    print(i, '*', a, '=', i * a)
```

- we need to add command for printing the "line"
- it is going to be repeated in each line with numbers => it is supposed to be part of the loop
- we add it with the same indentation as the command that prints the numbers has

By setting the same indention of the commands in the loop, we say that all of them are supposed to be repeated within one step of the loop - the loop will go to the next step only when it executes all commands with the same indentation.

```
entry = input('Enter a number: ')
a = int(entry)
for i in 1,2,3,4,5,6,7,8,9,10:
    print(i, '*', a, '=', i * a)
    print('--------------------------')
```

- wrong entries:

```
for i in 1,2,3,4,5,6,7,8,9,10:
    print(i, '*', a, '=', i * a)
print('--------------------------')

# ends with a list of all multiplications and adds a "line" at
the end.

for i in 1,2,3,4,5,6,7,8,9,10:
    print(i, '*', a, '=', i * a)
  print('--------------------------')

# will result in an error because it is not possible to
determine which part of the program the second print()
statement belongs to.
```

7.2 Range of values

7.2.1

- writing the list of values in a loop, especially if they are consecutive, is pointless
- Python uses the **range()** function to generate the list of numbers

- the simplest version of **range()** - entering a single parameter - the number of integers (how many numbers from zero will be in the list)
- For example:

```
range(3)
```

It returns the list of values - numbers:

```
0, 1, 2
```

- using the command itself does nothing
- the sequence is created that we can use directly in the loop

```
for i in range(3):
    print(i)
```

- for repeating the action **n-times** it is enough to generate the list with **n** values using **range()**
- generated list always **begins with the value 0** and ends with the value n-1

```
n = 5
for i in range(n):
  print(i)
```

7.2.2

- to generate the list of values that does not start with zero, we add second parameter to the **range()** function

```
range(start, stop)
```

- **start** determines the starting element of the sequence
- **stop** determines the final element of the sequence

```
z = range(10, 15)
for i in z:
    print(i)
```

Prints the sequence:

```
10, 11, 12, 13, 14
```

7.2.3

Complete the parameter of the **range()** function so that the following list is displayed:

```
15, 16, 17, 18, 19, 20
```

```
for i in range(15,21):
    print(i)
```

Program output:

```
10
```

7.2.4

- in **range()** it is possible to define a step by witch the values in the list will change by leaps

```
range(start, stop, step)
```

- the leap of the change is determined by the third parameter - **step**

```
for i in range(10, 31, 5):
    print(i)
```

- for the parameters **start** and **stop** applies the same rules
- the last value of the sequence is at least 1 less than the **stop** value

7.2.5

Which values can be stored in the variable **end** to generate a list

```
5, 9, 13, 17, 21
```

```
end = ?
z = range(5, end, 4)
for i in z:
  print(i)
```

7.2.6

Step parameter can also have **negative values**, that can be used to create a list with descending values.

```
range(20, 15, -1)
```

Creates:

```
20, 19, 18, 17, 16
```

- for values **start** and **stop** in this case has to apply star > stop
- value given in **stop** does not display in the list

```
z = range(20, 10, -3)
for i in z:
  print(i)
```

Creates:

```
20, 17, 14, 11
```

7.2.7

How many values will the list generated by the **range()** function have?

```
z = range(20, 0, -5)
for i in z:
  print(i)
```

Operations in a Cycle

Chapter 8

8.1 Revision

8.1.1

Management structures

- **sequence** - the progression of commands

```
tradius = input("Enter radius: ")
radius = int(tradius)
print('area:', 3.14 * radius * radius)
print('circuit:', 3.14 * 2 * radius)
```

- **branching** - part of the program is only executed if the condition is met

```
tsalary = input('Enter the salary of the potential spouse: ')
salary = int(tsalary)
if salary < 500:
  print('run away quickly')
elif salary < 1000:
  print('think about it')
elif plat < 2000:
  print('as long as there are two of you, you can make a
living out of it')
else:
  print('he is the one')
```

- **cycle** - repetition of the part of the program

```
tnumber = input("Enter a value: ")
number = int(tnumber)
for i in range(number):
  print('hi')
```

8.1.2

Definition of repetition

We define the number of repetition:

by listing specific values as part of the line defining the cycle

- by creating some **list of values**, e.g. using **range()**

```
for i in 1,2,5,8,7:
  print(i)
```

Or:

```
for i in 'mum','dad','grandpa','grandma','brother-1','brother-
2':
  print(i)
```

By using **range()** that creates the list:

for 1-parameter assignment - **range(n)** - <0,n-1) values are generated

```
for i in range(10):
  print(i)
```

- for 2-parameter assignment - **range(a,b)** - <a,b-1) values are generated

```
for i in range(10,15):
  print(i)
```

- for 3-parameter assignment - **range(a,b,step)** - <a,b-1) values are generated but in such a way that the **step** is added to the generated value

```
for i in range(10,20,3):
  print(i)
```

Program output:

```
10
13
16
19
```

8.1.3

Comments

- programs are becoming more extensive
- so far we have met with comments that were telling us what is supposed to happen in the program

```
# data loading

# calculation

# data output
```

- such **comments** serve for later understanding of the code or for another programmer
- comment is not intended for the program interpreter and is ignored by the interpreter
- comment star with a # character and the compiler ignores the text from this character to the right until the end of the line
- informal rule is that behing the # character there is a space

```
txt = input('enter an input') # this line reads the radius
r = int(txt) # cast to an integer
```

- a multi-line - **block** comment can be also used
- it begins and ends with three quotation marks or apostrophes

```
"""This is a block comment
        it can contain several lines."""
print(10+20)
```

8.2 Calculations in a cycle

8.2.1

A list of income for the whole year is given. Find out how much you had in your bank account for each month, ussuming you did not spend anything of it.

- the amounts of money are: **1000, 1200, 1120, 1450, 1300, 1210, 1800, 1500, 1354, 1290, 1358, 1700, 1882, 1730**
- it is a list for which we have to know the current value after adding the sum
- at the beginning we will probably have zero in our account

```
amount = 0
```

- the list of values changes every month, the money amount increases by the given value
- the operation repeats = we use cycle
- it always adds the current value = we lists the amounts to be added

```
for i in 1000, 1200, 1120, 1450, 1300, 1210, 1800, 1500, 1354,
1290, 1358, 1700, 1882, 1730:
  amount = amount + i
print(amount)
```

- we also have a request to print the current value, so we add:

```
amount = 0
```

```
for i in 1000, 1200, 1120, 1450, 1300, 1210, 1800, 1500, 1354,
1290, 1358, 1700, 1882, 1730:
  amount = amount + i
  print(amount)
print('total amount: ', amount)
```

- or with details:

```
amount = 0
for i in 1000, 1200, 1120, 1450, 1300, 1210, 1800, 1500, 1354,
1290, 1358, 1700, 1882, 1730:
  print('amount =',amount,'+',i,'=',amount + i)
  sum = sum + i
print('total amount: ', amount)
```

8.2.2

Calculate the sum of first 100 positive numbers.

- we need to sum up the values 1 + 2 + 3 + 4 + 5 + 6 ... + 99 + 100
- we will add them gradually, in a cycle, which will be repeated from 1 to 100
- in every step we add i-value
- we use _sum variable for storing the temporary result
- sum is empty on the beginning - it containts the value 0

```
sum = 0
for i in range(1, 101):  # the last value is supposed to be
100
    sum = sum + i
print(sum)
```

- or with output:

```
sum = 0
for i in range(1, 101):  # the last value is supposed to be
100
    print(i,'  sum = sum +', i, ' i.e.', sum, '+', i, '=', sum
+ i)
    sum = sum + i
print('sum is:',sum)
```

8.2.3

Calculate the product of first n numbers given n as an input.

Loading is simple:

```
t_n = input()
n = int(t_n)
```

- e.g., for 10 it is a multiplication of 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10
- we generally have to execute it sequentially just like addition

If we break down the individual steps of the calculation, we need some "storage" with a neutral value:

```
product = 1
```

- in case of multiplication it is the value 1 - the result of the calculation does not change, if we used the value 0 for multiplication, the result would always be 0
- in case of sum it is the value 0 - the result would not change by adding 0

```
product = 1
product = product * 2   # we multiply the original value by 2
- 1*2=2
product = product * 3   # we add the multiplication by 3 -
2*3=6
product = product * 4   # we add the multiplication by 4 -
6*4=24...
```

- or:

```
product = 1
for i in range(1, n + 1):
    product = product * i

print(product)
```

- or the entire program:

```
t_n = input()
n = int(t_n)
product = 1
for i in range(1, n + 1):
    print(i,'  product = product *', i, ' t.j.', product, '*',
i, '=', product * i)
    product = product * i

print('result:',product)
```

8.2.4

Multiply the enterd sequence of numbers. Display the number that is being multiplied at the moment.

E.g., the printout for the sequence: 4, 9, 2, 5, 2 looks like:

```
1 - 4
2 - 36
3 - 72
4 - 360
5 - 720
```

- for displaying the serial number we need the counter
- at the beginning we have the value of zero - no step has been executed yet
- in every step of the cycle the value increases by 1
- the variable will be called **step**

```
product = 1          # we set the neutral value for the
multiplication
step = 0           # we will count how many time has the cycle
been executed
for i in 4, 9, 2, 5, 2:
   step = step + 1     # the cycle was executed again,
increase the step by 1
   product = product * i   # we multiply
   print(step, '-', product)   # we print it
```

8.2.5

Write a program that finds out how much have you spent for your purchases in the last week. It asks for the daily amount of money spent.

- we do not know the amounts spent in individual days, we only know that there is 7 days
- the code can look like:

```
_sum = 0
text = input('Enter the amount: ')      # load the amount for
the first day
daily_consumption = int(text)        # convert it to a number
_sum = _sum + daily_consumption    # add the amount for the
first day

text = input('Enter the amount: ')      # we add the amount for
the second day
```

```
daily_consumption = int(text)        # cast it to a number
_sum = _sum + daily_consumption    # add the amount for the
second day

text = input('Enter the amount: ')     # load the amount for
the third day
daily_amount = int(text)        # convert it to a number
_sum = _sum + daily_consumption    # add the amount for the
third day
```

What is being repeated?

- **more commands**(steps) - we can repeat the arbitrary number of commands in the cycle
- we use the same variable for loading the values from the user (**text** and **daily_consumption**)
- the variable can be used for storing the value more than one time
- the old value is always overwritten by a new one

This sequence of commands:

```
text = input('Enter the amount: ')     # load the amount
daily_amount = int(text)        # cast to a number
_sum = _sum + daily_amount    # add the amount
```

is repeated for 7 times.

After the loading and adding of all the amounts to the variable _sum the program prints only the result:

```
_sum = 0
for i in range(7):         # week has seven days, it is
repeated for 7 times
    text = input('Enter the amount : ')
    daily_consumption = int(text)
    _sum = _sum + daily_consumption

print('In the week you spent ', sum
```

8.2.6

Get the precipitation for the last n days and calculate their average value. The values are eneter in whole milimetres.

- we do not know anything at the beginning - not the rainfall values, not their number
- at first we need to know/read the number to be used as a number of repetition of the loop

```
text = input('Enter the number of days: ')
n = int(text)
```

- we can get the average as sum/number
- we load the number of values at the beginning of the program
- we will continuously oncrease the sum by loading the precipitation on individual days

```
sum_ = 0
for i in range(n):
    text = input('Enter precipitation : ')
    z = int(text)
    sum_ = sum_ + z
```

- to find out the average we divide the sum by the number - the "/" operator is used for the division

```
print('Average value of precipitation in', n, 'days is',
_sum/n)
```

The result is the number with a decimal point, e.g.:

```
5.0
```

even for integers. That is how Python inform us that the result is generally a real number.

Data Types

Chapter 9

9.1 Data types

9.1.1

Every variable is defined by two things:

- **title** (name), according to which we refer to it
- **data type**, which determines whether it stores text or a number

Experience:

data loading returns input in text form

- if we wanted to work with such data as with number, we had to **retype** them

```
t_n = input('Enter number')
n = int(t_n)
```

- **Python** is a language with dynamic type checking - it is not necessary to define data types for variables in advance
- the type of the variable is based on the values

Most commonly used data types are:

- **integer** - integers, e.g. 10 or 5
- **float** - floating point numbers - storing decimal numbers, e.g. 10.5 or 5.3
- **string** - a string represented by apostrophes or double quotes

When writing decimal numbers we use decimal point - a comma is evaluated as an error.

```
x = 4     # 4 - will be interpreted as an integer, data type
integer
y = 4.0   # 4.0 - will be interpreted as a number with a
decimal point, data type float
z = "4"   # "4" - will be interpreted as a string, data type
string
```

9.1.2

Type

- for every variable we can find out its type using the **type()** function

```
a = 4
print(type(a))
```

Program output:

```
```

- the printout means that the entered value is treated as an integer

Likewise:

```
b = "Python"
print(type(b))
```

Program output:

```
```

- it means that the entered value is treated as a string

```
c = 4.2
print(type(c))
```

Program output:

```
```

- it represents a decimal number

9.1.3

Casting functions

- the acquisition of a value by a variable determines the data type of a variable
- to make it possible to execute operations with the concents of variable intended for different data type, we need operations that can transform data between individual types

Python supports three basic casting functions, that convert one data type to another data type

- **int()** - returns a value with a data type **int**
- **str()** - returns a value with a data type **str**
- **float()** - returns a value with a data type **float**

```
a = int(10.5)       # returns 10
b = int('69')       # returns 69
c = float(100)      # returns 100.0
d = float('35.53') # returns 35.53
e = str(24)         # returns '24'
f = str(66.99)      # returns '66.99'
```

- if the cast operation fails, the compiler display an error message, e.g.:

```
aa = 'text'
a = int(aa)
```

Program output:

```
ValueError
invalid literal for int() with base 10: 'text'
```

9.1.4

Get the result of the substraction of two values stored in integer variables and display it in a form:

```
8-5=3
```

- using **print()** and separating parts with commas, we do not get required outcome, because it inserts spaces between the parameters
- we combine numbers and text, converting numbers to a text
- we use **str()** function, which can be use both for a variable and for the result of a mathematical operation

```
a = 8
b = 5
result = str(a) + '-' + str(b) + '=' + str(a - b)
print(result)
```

Program output:

```

```

9.2 Integers

9.2.1

Numeric types

Numeric type variables (int and float) store numeric values and allow us to execute following operations:

- + (sum) a + b, e.g.: 10 + 3 = 13
- - (difference) a – b, e.g.: 10 - 3 = 7
- * (product) a * b, e.g.: 10 * 3 = 30
- / (share) a / b, e.g.: 10 / 4 = 2.5

- // (integer share) a **//** b, e.g.: 10 // 3 = 3, while the decimal part is neglected

- % (remainder after integer division) a **%** b, e.g.: 10 % 3 = 1
- ** (power) a ** b, e.g. 5 ** 3 = 125, so 5 * 5 * 5

9.2.2

Remainder after division

- integers have an operation that returns a remainder when divided
- % operator is used for notation
- in the last case, an error will occur - you cannot divide by zero...

```
print(10 % 3) # result - 1
print(10 % 2) # result - 0
print(15 % 4) # result - 3
print(20 % 7) # result - 6
print(10 % 0) # ZeroDivisionError: integer division or modulo
by zero
```

Program output:

```
1
0
3
6
ZeroDivisionError
integer division or modulo by zero
```

What is going to be stored in variable **x** after the program ends?

```
y = 22
z = 4
x = (y // z) ** (y % z)
print(x)
```

Program output:

```
25
```

9.2.3

Evenness and oddness

- using the modulo (%) operation we can define, whether the entered number is even or odd
- for odd numbers, the remainder after the division with 2 is 1
- in case of even numbers is 0

condition, that finds out whether the number is even or odd, can look like:

```
n = 10
if (n % 2 == 0):
  print(n, 'is even.')
else:
  print(n, 'is odd.')
```

- or:

```
n = 10
if (n % 2 == 1):
  print(n, 'is odd.')
else:
  print(n, 'is even.')
```

9.2.4

Negative numbers

- they represent exactly half of all registrable integer values
- we enter a negative number using the - sign before the numerical value

```
c = -1
d = 15 + -5
```

- we can enclose a negative value in parentheses:

```
e = 15 // (-5)
```

9.2.5

Abbreviated entry

- Python provides the special shorthand version for changing the value of a variable
- it is used in entries when we want to change the original value of a variable by an arbitrary operation

E.g., the entry:

```
num = num + 1
```

Can be abbreviated to:

```
num += 1
```

- the abbreviated entry could be read as increasing the value of the **num** variable by one.

There must be no space between the operation sign and the assignment sign (=).

Similarly, the entry:

```
num = num - 3
```

is equivalent to entry:

```
num -= 3
```

- we change the value of the variable poc by subtracting the value 3 from it

As well as addition and subtraction, we can also enter other operations.

```
# Increase the value in variable x by a factor of ten.
x = x * 10
x *= 10

# Decrease the value of b by 15.
b = b  - 15
b -= 15

# Halves the value stored in the amount variable.
_sum = _sum  / 2
_sum /= 2

# Execute an integer division of the contents of the sum
variable by three.
_sum = _sum  // 3
_sum //= 3

# Power the contents of the number variable to the fourth.
number = number ** 4
number **= 4
```

9.2.6

What is going to be stored in the variable **cnt** after the execution of the following steps?

```
cnt = 3
cnt *= 5
cnt //= 4
```

```
cnt -= 5
cnt += 15
cnt //= 6
cnt **= 3
print(cnt)
```

9.2.7

- in abbreviated entry, we do not have to display only a numerical value on the right side, we can also use a variable

Entry:

```
a += b
```

means that the value of the variable **a** will be increased be the contents of variable **b**.

What is the following program going to print?

```
a = 3
b = 8
a *= b
print(a)
```

9.2.8

Write a program, that finds out and prints the number of divisors for a given number.

```
# ...
```

9.3 Decimal numbers

9.3.1

- Python uses data type float to store decimal (real) numbers
- the decimal part is separated from the integer **by a dot**

```
5.18
```

- we can recognize the real number in the printout by the fact it has **displayed decimal part**
- a number with zero decimal part displays it as .0

```
a = 10
b = 2
c = 10//2
d = 10/2
print(c) # the result of integer division is an integer
print(d) # the result od "classical" division is a real number
```

- if we want to specify that the given variable is supposed to be a **float** type we let the compiler know by inserting a decimal value

```
a = 12.5
b = 3.0    # in this case we also enter an integer as a
decimal number
```

9.3.2

Real numbers are entered in a standard format:

```
3.1415296536, 556.44
```

Or in scientific format:

```
5.5644e2
```

- which means **5,5644 * 10^2 = 556,44**
- in this notation the integer is between 1 and 10, while the shift of the decimal point is provided by the exponent

How much is:

```
328 = ??? # in scientific format
578.33 = ??? # in scientific format
0.328 = ??? # in scientific format
0.00328 = ??? # in scientific format
```

```
5.14e2 = ??? # in standard format
5.14e3 = ??? # in standard format
5.14e-1 = ??? # in standard format
```

9.3.3

- the combination of an integer and real number type or the operation for real numbers returns a real number as a result

The outcome of the following code...:

```
a = 10
b = 5
c = a / b
print(c)
```

Program output:

```

```

...is a real number.

9.3.4

Loading a decimal number

- we use **input()** to load a decimal number
- then we convert the text to a decimal number using the **float()** function

```
aa = input('Enter a number: ')
a = float(aa)
```

Determine which one of the two decimal numbers in the input is larger.

```
# ...
```

9.4 Mathematical functions

9.4.1

Mathematical functions

- each data type has a set of the standard functions by which we can process the values
- they are called the **buit-in functions**
- we do not have to add any "adds-on" to the program to be able to use these functions
- it applies, that the function processes the value or the values which we enter in the parentheses (we call them parameters or arguments)
- **it returns the result,** we can continue to work with, e.g. print it or insert it to a variable

Every data type has its own built-in functions. For example, numeric data types have these functions:

- **abs()** - returns the absolute value of the entered number
- **max()** - returns the maximal value of the entered values
- **min()** - returns the minimal value of the entered values

- **pow(x, y)** - returns x squared by y, it is the same calculation as x ** y

```
a = abs(-3)                # result 3
b = max(2, 5, 6, 8, 1, 3) # result 8
c = min(2, 5, 6, 8, 1, 3) # result 1
d = pow(3, 2)              # result 9
e = abs(-3.5)              # result 3.5
```

9.4.2

Rounding

- rounding has a special place when working with real numbers
- **the round()** function processes decimal value by rounding it to a integer value

```
round(3.45) # result 3
round(5.75) # result 6
round(-1.6) # result -2
```

In **Python**, .5 values are rounded:

down in case there is an even number before .5, e.g.:

```
round(4.5) # result 4
round(6.5) # result 6
```

- up in case there is an odd number before .5, e.g.:

```
round(5.5) # result 6
round(7.5) # result 8
```

What is the result of the following program?

```
a = 14
b = round(a/4*100)/100
c = a - b
d = round(c)
print(d)
```

9.4.3

Rounding 2

- for rounding to the specified number of decimal places is used the prevalent version of the **round()** function
- the second parameter defines the number of decimal places to which the value is supposed to be rounded

E.g., to round to two decimal places:

```
pi = 3.14159
pi2 = round(pi, 2)
print(pi2)
```

Program output:

```

```

In case of entering the negative value the value is rounded to tens, hundreds, etc. For example:

```
a = round(1234, -1)  # result 1230
b = round(1234, -2)  # result 1200
c = round(1254, -2)  # result 1300
d = round(2854, -3)  # result 3000
```

What is the result of?

```
print(round(13.67, 1)) # result
print(round(1.865, 2)) # result
print(round(136, -1)) # result
print(round(136, -2)) # result
```

9.4.4

Function nesting

- each function processes the arguments and returns the result, which we can continue using, e.g.:

```
x = max(10,20)
print(x)
```

- we can merge the entry into one command
- we leave out the assignment of the result of the function max() to a variable and simply print the result

```
print(max(10,20))
```

The compiler evaluates the entry in this way:

firstly it processes the innermost function - the maximum of 10 and 20

- it returns the result
- uses it as an argument for the **print** function

We can also use:

```
a = round(max(5.6, 7.8))
```

- the innermost **max()** function is evaluated first
- its result becomes the argument of the round() function
- then the result is assigned to the variable **a**

This way we can nest practically any number of functions in to each other.

What is the result for the following sequence of commands?

```
a = 10
b = 3.6
c = min(a, round(abs(round(b-a)) * 3))
print(c)
```

9.4.5

One-line data loading

Just like we can nest mathematical functions, we can also nest and process all other functions.

Due to they textural form, we had to carry out the loading of numerical data from the user in two steps:

- loading
- conversion to a number

```
aa = input()
a = int(aa)
```

This procedure can be simplified thanks to nesting the input() function into the int() function that transform text into a number.

```
a = int(input('Enter integer number'))
```

Or:

```
b = float(input('Enter decimal number'))
```

Write a program that reads an integer and a decimal number from the input. Integer represents the liters of fuel in a car's tank and the decimal number informs the program about the fuel consumption per 100 km.

- **Subsequently, the program displays how many full kilometers the fuel in the tank is sufficient for.**

```
# ...
```

9.5 Boolean expressions

9.5.1

Boolean expression

video

We are familiar with the expressions:

```
x = 1 + 2 * 3
a = '10'
b = '20'
c = a + b
```

- in addition to expressions whose result is a number, we often work with expressions whose result is **true** or **false**
- we refer to these expressions as **boolean** and we have already encountred them when using conditions in the **if** statement

we often compered variables with each other or variables with values, we can also compere the values themselves, e.g.:

```
if 4 > 0:
    print ('true')
else:
    print('false')
```

- we can replace this code with just the output:

```
print(4 > 0)  # prints True
```

- if the expression in the brackets is true then its result is **True** value, in opposite case the result is **False**

is it true that 5 - 3 < 0?

```
print(5 - 3 < 0)  # prints False
```

Pay attention to the size of the letters. The values **true** or **false** are not boolean values.

Complete the code to determine whether the n is positive.

```
txt = input('Enter a number')
n = int(txt)
print(n    )
```

9.5.2

Bool type

video

- to store the boolean values we use **boolean** (**bool**) type variables
- therefore we usually obtain their content as a result of comparison, verification of the truth of the condition, etc.

The condition, whether **a > b**, can be verified by the entry in the **if** structure:

```
if a > b:
```

Alternatively, we can also store the result of the evaluation into a variable:

```
a = 10
b = 5
result = a > b
print(result)
```

- if the value **a** is larger than the value **b**, the value stored in the **result** variable will be **True**
- in the opposite case (lesser or equal), the **result** variable will contain the **False** value after the evaluation
- we can also check the data type of the resulting value:

```
print(type(result)) # prints
```

Program output:

9.5.3

Boolean variable

video

- if we can print the evaluation of the expression, we can also store it in a variable:

```
result = 4 > 0
print(result)
```

- the code first evaluates whether 4 > 0
- then the **True** value corresponding to the truth is stored into the **result** variable

- in another code it is firstly evaluated whether **4 > a + 5**, therefore **15**
- then the False value corresponding to the untruth is store into the **result** variable

```
a = 10
result = 4 > a + 5
print(result)
```

9.5.4

Comparison operators

video

- **>** - is larger, e.g. **a > b**
- **>=** - is larger or equal, e.g. **a >= b**
- **<** - is lesser, e.g. **a < b**
- **<=** - is lesser or equal, e.g. **a <= b**
- **==** - is equal, e.g. **a == b**
- **!=** - is not equal, eg. **a != b**

Using characters in the wrong order will cause an error (e.g.: =>, or <>).

```
a = 10
b = 7
result = a => b
```

Program output:

```
```

9.5.5

The result of comparison can be also used on the conditions:

- at first we find the result of the expression
- then we use it in a condition, e.g.:

```
a = 10
b = 5
result = a == b
if result == True:
  print("The values are equal")
else:
  print("The values are different")
```

Entry...:

```
if result == True
```

...we usually enter in the form:

```
if result
```

- because the evaluation of the condition **result == True** depends on what value the **result** variable has
- if the condition is true

```
if result == True
```

- we ask whether the truth is truth (**True == True**) – the result is **True**

if the variable contains the value expressing false, then the condition is still the same:

```
if result == True
```

- we ask whether the false is true (**False == True**) – the result is **False**.

The answer to the condition is actually already contained in the **result** variable:

- if it contains the true value, the result is true (**True**)
- if false, the result is **False**

So the following entry is correct and efficient:

```
if result
```

9.5.6

Compound conditions

video

In program, we often combine multiple conditions which can be in a different relationship.

Most often, we encounter the situations in which:

- all conditions must apply simultaneously
- it is enough for one of the conditions to apply

According to the entered age of the employee find out whether he is in the productive age - between 18 and 70 years.

Task can be solved as follows:

```
age = int(input('Enter the employee´s age: '))
if age >= 18:        # whether the first condition is met
  # check, whether the age is also simultaneously less then
the upper limit
  if age <= 70:      # both conditions are met
    print("The employee is in the productive age")
```

Or we can combine the conditions into a single expression and evaluate them trough a single **if**:

```
age = int(input('Enter the employee´s age: '))
if (age >= 18) and (age <= 70):
    print("The employee is in the productive age")
```

- the notation allows to enter two verifications into a single **multiple condition**
- the fact that they are supposed to apply simultaneously is expressed by using the boolean clutch **and** (and simultaneously)
- in a multiple condition, complete conditions are combined - i.e. **variable must be specified in each subcondition**

Python does not require the use of parentheses when creating a multiple condition, but we recommend them to avoid various errors.

9.5.7

Condition improvement

- Python allows to delimit a variable with comparison operators on both sides
- this way we can abbreviate the notations of the boolean expressions
- we do so only if the same variable occurs in both conditions

The expression...:

```
(n >= 0) and (n <= 10)
```

...can also be entered as:

```
0 <= n <= 10
```

The entry of our program will take on a new form:

```
age = int(input('Enter the employee´s age: '))
if (18 <= age <= 70):
    print("The employee is in the productive age")
```

Or:

```
age = int(input('Enter the employee´s age: '))
if (70 >= age >= 18):
    print("The employee is in the productive age")
```

9.5.8

Or command

video

- in some cases, the fulfillment of the one of the verified conditions is sufficient
- we use boolean clutch **or**

```
if (a > 0) or (b < 0)
```

The evaluation of the expression is true:

- if at least one of the conditions is met – i.e. it is sufficient if **a > 0** or **b < 0**
- if both conditions are met

Complete the program so it prints that the number is accepted in case it is negative or even.

```
number = int(input('Enter a number: '))
if ():
    print('accepted')
```

9.5.9

Compound expressions

video

- simultaneous validity of the several conditions - **and**,
- fulfillment of **at least** one condition from the group - **or**.

The use of this pair is not limited for the use in **if**,

- we can use them while working with boolean expressions
- E.g. the result of the expression stored in the **c** variable

```
a = 10
b = 5
c = a > b or b < 0
print(c)
```

- obtained by successively evaluating the individual parts of the compound condition
- at first we evaluate each part separately:

```
c = a > b  or  b < 0
    10 > 5 or  5 < 0
    True   or  False
```

- the result of the combination of the truth values true (**True**) or (**or**) false (**False**) is **True**.

9.5.10

Let's test the requirement for the number to be both positive and even:

```
n = 15
```

We test whether the number is both positive and even:

```
n > 0         # the result is True
n % 2 == 0    # the result is False
```

We connect the results with the boolean clutch **and**:

```
result = (n > 0) and (n % 2 == 0)
              True and False
                  False
```

9.5.11

Negation

In addition to checking whether the condition is true, we can also use the notation: if it is not true, then, e.g.:

```
a = 5
b = 1
zeroDevisor = b == 0  # in this case the result is False
if not(zeroDevisor):  # if it is not true that the devisor
equals zero
  quotient = a / b
  print(quotient)
else:
  print("Attempting to divide by zero")
```

- an entry starting with the expression **not** - **negates** the result of the expression or the content of the variable listed after it
- it makes **False** from the **True** value and vice versa

In this case the **zeroDivisor** contains the **False** value and the notation in the condition means:

- if it is not true, that **zeroDivisor**, calculate the quotient
- resp. if **zeroDivisor** contains the **False** value, then execute
- resp. if negated content of the **zeroDivisor** variable is true, then execute

9.5.12

Priority of operations

- when evaluating, we need to take to account the priority of individual operators
- we know from mathematics that * a / takes precedence over + a -
- in Python, the operations listed in the line above are executed first

```
*, @, /, //, %
+, -
in, not in, is, is not, <, <=, >, >=, !=, ==
not x
and
```

```
or
```

9.5.13

Compound expressions

The combination of the boolean expressions and boolean variables does not need to be limited to only two elements

E.g.:

```
h1 = False
a = 5
b = 7
result = not(a > b) or (b - 5 < a) and h1 or not(h1)
print(result)
```

...will be evaluated with respected priorities such as:

```
not(a > b) or (b - 5 < a) and   h1     or  not(h1)
not(False) or     True    and   False  or  not(False)
    True   or     True    and   False  or  True
    True   or             False        or  True
                  True                 or  True
                                      True
```

What is the result of:

```
k = 4
j = 2
result = (k >= 5) or (j >= 6) and (j >= k)
print(result)
```

Program output:

```
False
```

How to adjust the following expression so that the result is True?

```
k = 83
j = 15
result = (k % 20 >= 5) or (j <= 6) and (j < k)
print(result)
```

Strings

Chapter 10

10.1 String

10.1.1

String

- data type
- allows us to store and work with a group of characters
- it usually represents a word or continuous text

Enclosement:

- apostrophes (')
- quotation marks(")

E.g.:

```
name = 'Adam'
```

...inserts the **Adam** text content into the **name** variable.

For entry...:

```
name = Adam
```

...the compiler would expect the variable called **Adam**, whose content it would insert into the **name** variable.

10.1.2

String operations:

- connecting - the simplest operation
- the execution provides the "+" character
- it creates the third string form the two string by connecting the content of the second string to the end of the content of the first one

```
result = 'attention' + 'bites'
print(result) # prints attentionbites
```

We can connect any number of strings or variables, that contain string.

```
a = 'Mum'
b = 'has'
c = 'Ema'
d = a + b + c
print(d) # does it print Mum has Ema?
```

10.1.3

String and number connecting

- we need to use number to string casting using **str()** function

```
s = 'result: '    # text
a = 3             # number
b = 7             # number
c = a + b         # number
d = s + str(c)
print(d)
```

- if we wanted to use "+" sign to connect a text and a number we would get an error message

E.g.:

```
a = "result: "    # text
b = 3             # number
c = a + b
c = b + a
```

10.1.4

String

- string is often used to indicate the type of a variable containing a text string
- **type(variable)** returns **str** value

```
a = "Python"
print(type(a))
```

Program output:

```

```

10.1.5

Casting at the input

- when loading data, all data enter the program as text strings
- for further processing in a different way, it is necessary to cast them

```
print('Enter an integer: ');
entry1 = input()
a = int(entry1)                          # cast to an integer
```

```
print('Enter a decimal number: ');
entry2 = input()
b = float(entry2)                    # cast to a decimal number
c = a - b
result = 'Difference: ' + str(c)      # cast the number to a
text
print(result)
```

10.1.6

String multiplication

- in addition to summation, Python also allows to use string multiplication
- one variable needs to be a string and the other on an integer, e.g.:

```
n = 3
txt = 'uff '
c = n * txt
print(c)
```

- the result of the multiplication is a text string containing the content of the text variable repeated n-times in a r
- operations such as division or substraction can not be used on strings

10.1.7

What is the result of the following steps?

```
ab = '10'
print(ab + ab)
print(ab + 'ab')
print('ab' + 'ab')
print('ab' * 3)
print(ab * 3)
print(int(ab) * 3)
print(int(ab) + 3)
print(ab + 3)
print(ab + str(3))
```

10.2 The basics of working with a string

10.2.1

Working with text

- finding the length of the text
- iterating trough characters
- searching for character or string
- changing the size of characters, etc.

10.2.2

String length

- = number of characters
- **len()** function

```
data = "Mum"
length = len(data)
print(length)
```

- the number of the characters contained in the **data** variable was stored in the **length** variable

What does the following code print?

```
print(len(''))
```

10.2.3

Index

- string consist of characters
- each character has its place in the string defined by an **index**
- index of the first character is **0** => the second at position 1 etc.
- the last character is on the position one less than the total number of characters in the string

E.g. for:

```
data = "Madonna";
```

the characters are distributed at individual positions as follows:

index / position	0	1	2	3	4	5	6
character	M	a	d	o	n	n	a

The number of characters in the string 7, the last character is in position 6.

10.2.4

Characters

- access to characters is enabled by the notation consisting of the variable name followed by square brackets with an index

E.g.:

```
data = 'Sit down'
first = data[0]
print(first)
```

Access to individual characters:

```
data = 'Sit down'
first = data[0]
print(first)   # prints S
print(data[1]) # prints i
print(data[2]) # prints t
print(data[3]) # prints space
print(data[4]) # prints d
print(data[5]) # prints o
print(data[6]) # prints w
print(data[7]) # prints n
print(data[8]) # prints an error
```

- the last character of the string has an index one less than the length of the entire string, i.e.:

```
last = len(data) - 1
```

10.2.5

Iteration trough characters of a string

- we can access each character in the string trough its index
- if we need to iterate trough all of the characters, we usually do this by using the loop

Print the entered word letter by letter below.

```
data = input('Enter a string: ')
length = len(data)
for i in range(length):
    print(data[i])
```

- **range(length)** generates the values starting with zero and ending with the **length-1** value
- from the index of the first to the index of the last character

10.2.6

Write a program that prints letters in even positions of the string (characters in position 0, 2, 4, 6, ...).

- we generate the list of values using range
- starting with zero
- ending with the length of the string
- with step 2, therefore 0, 2, 4...

```
data = input('Enter a string: ')
length = len(data)
for i in range(0, length, 2):
  print(data[i])
```

10.2.7

- we can use the **for** loop feature, where instead of generating a range we can directly enter a string

```
data = 'Slovakia'
for i in data:
  print(i)
```

- at each step of the loop, the next character in the specified string is inserted into a variable i
- we use iteration when we want to process only the characters of the string, but **we are not interested in their position**

10.3 Typical tasks

10.3.1

Write a program that finds how many times the character 's' is found in the given word.

- we iterate trough the word - trough characters
- if we find the 's' character, we count it in

```
word = input('Enter a word: ')  # read the word from the input
length = len(word)              # find the number of
characters
# the number of found characters will be stored in the number
variable - 0 at the beginning
number = 0
for i in range(length):           # we iterate trough the
positions in loop from 0 to lenhgt-1
   if word[i] == 's':             # if there is an 's' in the
position i, we increase the number
       number = number + 1
print('The word contains', number, 'of "s" characters') #
output
```

10.3.2

- the same thing once again

Write a program that finds how many times the character 's' is found in the entered word.

- and the solution is only possible in Python
- if we use strig in **in**, the loop iterates it trough the characters
- it inserts the following character into the variable i at each step:

```
word = input('Enter a word: ')  # read the word from the input
number = 0                      # number of found characters -
0 at the beginning
for i in word:                  # we iterate trough the
characters in the loop
    if i == 's':                # if the investigated
character is 's', we increase the number
        number = number + 1
print('The word contains', number, 'of "s" characters') #
output
```

10.3.3

Working with numbers

Write a program that finds out how many even and odd digits are present in the number read from the input as a string .

- we will work with characters as with numbers - digits (125 has digits 1, 2, 5)
- we need to cast each character to a number using **int()**
- then using % we determine, whether the digit is even

```
number = input('Enter a number: ')  # we read the number from
the input
even = 0                            # the number of even and
odd is 0
odd = 0
for i in number:                    # in the loop we iterate
trough the digits in the word
   digit = ?                        # each character (digit)
is casted to a number
   if digit ??? :                   # aif the digit is
divisible by 2, we increase the even
      even = even + 1
   else:                            # else odd
      odd = odd + 1
print('The number contains', even, 'even and', odd, 'odd
digits.')
```

10.3.4

And again the same thing:

Write a program that finds out how many even and odd digits are present in the number read from the input as a string.

- we do not have to solve everything using mathematics
- we know that the digits, or characters 0,2,4,6,8 are even and the others are odd
- so if the read digit is 0 **or** 2, **or** 4, **or** 6, **or** 8, we increase the number of even otherwise odd
- if we remember the boolean expressions, we know that we write **or** using **or**
- the name of the variable is stated in each condition

```
number = input('Enter a number: ')  # we read the number from
the input
even = 0                            # the number of even and
odd is 0
odd = 0
for i in number:
    # if the investigated character is 0,2,4,6,8 in other
words 0 or 2 or...
    if i == '0' or i == '2' .... ??? :
```

```
        even = even + 1           # we increase the number of
even
    else:                         # else odd
        odd = odd + 1
print('The number contains', even, 'even and', odd, 'odd
digits.') # output
```

10.3.5

And the last improvement:

Write a program that finds out how many even and odd digits are present in the number read from the input as a string.

- again usable specifically in Python only

Entry:

```
if i in '0' or i == '2' or i == '4' or i == '6' or i == '8'
```

- we test, whether the variable has one or the other or another value
- we can use **in** operator, that tests, whether the value is present in the list>

```
if i in '02468':
```

The whole program looks like:

```
number = input('Enter a number: ')  # we read the number from
the input
even = 0                            # the number of even and
odd is 0
odd = 0
for i in number:
    if i in '02468' :
        even = even + 1           # we increase the number of
even
    else:                         # else odd
        odd = odd + 1
print('The number contains', even, 'even and', odd, 'odd
digits.') # output
```

Program output:

```
Enter a number:  1224The number contains 3 even and 1 odd
digits.
```

10.3.6

Similar program:

Write a program that finds out how many vowels there are in the entered word. Consider only lowercase letters.

- vowels are a, e, i, o, u, y
- we take each character from the read string and determine, whether it is present in this list (in)

```
word = input('Enter a word: ')     # read the word from the
input
vowels = 0                         # the number of vowels is
zero in the beginning
for i in word:                     # we iterate trough the
characters of the word in the loop
    if i in 'aeiouy':              # if the investigated
character is in the list of vowels
        vowels = vowels + 1        # increase their number
print('The word contains', vowels, 'vowels.') # output
```

10.3.7

Write a program that prints a mirror image of the entered word.

- Mama -> amaM
- winter -> retniw
- at first we create a separate variable into which we will insert characters
- we insert each following character in the sequence before the existing string, for exampl,. for the word Aladin we will proceed as follows:

```
A - result = "A"
l - result = "l" + result, i.e. "lA"
a - result = "a" + result, i.e. "alA"
```

- we simply rewrite the idea:

```
word = input()
mirror = ''
for character in word:
  # we insert the processed character before the string
created earlier
  mirror = character + mirror
  print(mirror)
print("result:", mirror)
```

10.4 Characters

10.4.1

Character

- the basic building element of a string
- we can compare characters based on their order, which resembles the alphabet

```
 'a' < 'b' < 'c' … < 'z'
```

- all uppercase letters are lesser than all lowercase letters

```
'A' < 'B' … < 'Z' < 'a' < … < 'z'
```

- it is the consequence of character encoding in the computer
- each character has its own numerical code based on which the system knows what form to give the character
- characters are usually represented by tan ASCII table containing 255 base characters
- currently the alphabets are encoded using Unicode/UTF-8 encoding, but the first 128 characters are encoded the same way
- not all characters are displayable, in printouts we only use characters from 32 to 126

```
print("ASCII table:")
for i in range(32,127):
  print(i, '-', chr(i))
```

10.4.2

Upper and lowercase letters

- the uppercase letter - between the first uppercase and the last uppercase character
- lowercase letter - between the first lowercase and the last lowercase character
- digit - between the smallest and the largest digit

```
letter = input('Enter a character: ')
if 'a' <= letter <= 'z':
   print('lowercase letter')
elif 'A' <= letter <= 'Z':
   print('uppercase letter')
else:
```

```
   print('it is not a letter')
```

10.4.3

Character and its code

- **ord()** –returns the position of the character in ASCII table, for example ord('A') return the value 65
- **chr()** – returns the character, which is in the entered position, e.g. chr(65) returns 'A'

For the entered character it prints its predecessor and successor.

- we use **ord()** and **chr()** functions
- we find the character's position in the ASCII table using **ord()**
- to get the predecessor, we use **chr()** to print a character at a position one smaller
- to get the successor, we use **chr()** to print a character one position larger

```
character = input('Enter a character: ')
pos = ord(character)
print('previous: ', chr(pos - 1))
print('next: ', chr(pos + 1))
```

10.4.4

What is the output of the code:

```
print(chr(ord('2') - 2))
```

10.4.5

Comparison

- we can determine the similarity of the strings trough a simple comparison

```
r1 = 'Grandma'
r2 = 'Grandpa'
if r1 == r2:
  print('the same')
else:
  print('different')
```

- string comparison does not tell which string is alphabetically larger or smaller

- for alphabetical comparison is used the "classic" comparison using character positions in the ASCII/Unicode table
- it progresses in both strings from the first position
- when different characters are encountered, their position in the ASCII table is compared
- the string, whose **first different** character is in the smaller position, is smaller than the other one

```
'Mother' > 'Father'
'Michal' > 'Michaela'
'Ivan' < 'Ivana'
```

10.4.6

Compare:

```
mum      dad
spring   summer
autumn   winter
Aladin   Jasmina
donkey   Shrek
```

10.4.7

Comparing numbers

- we usually compare numbers based on mathematical rules
- however, there may also be situations when we compare them lexicographically - as a text

It applies that:

```
0 < 1 < 2… < 9
'0' < '1' < '2'… < '9'
```

Although it applies that:

```
'12' < '13'
```

It also applies that:

```
'122' < '13'
```

- because the character '2' is at a smaller position in ASCII table than '3'.

10.4.8

What is the result of the following program?

```
a = 'Dingo'
b = 'Bingo'
print(a > b)
```

10.4.9

Find the maximum digit in the entered number. For example for 784541 it is 8.

- the number of digits is limited when using a numeric types
- we use string to load long numbers

The process is simple:

- at the beginning we declare the smallest possible value, therefore 0, as the largest digit
- we will gradually read the values from the individual positions of the string (from the beginning to the end) and compare them with the largest value found so far
- given that the alphabetical order od the digits is the same as their order by size, we can compare text

```
number = input()
mx = '0'           # or mx = number[0] we are working with
characters, so max will also be saved as a character
for digit in number:
  if mx < digit:   # if the current digit is larger than the
largest one so far
     mx = digit    # we will remember it
print(mx);
```

10.5 Special characters

10.5.1

Special characters 1

- they can be used for the printout of certain characters and adjusting the layout of the text
- \' – inserts an apostrophe into the text
- \" – inserts quotation marks into the text
- \\ – inserts a backslash into the text

```
print("We are starting to learn \"Python\"")
```

Program output:

```
We are starting to learn "Python"
```

- we used the quotation marks to enclose the string as well as in its content
- thanks to the writing in the form of \" there was no error

10.5.2

Special characters 2

- there are used for the printout formatting
- \n - new line character, moves the cursor to the beginning of a new line
- \t – tab, inserts a break, that indents the following text at the tab position

```
print('Hi, \nI am Emil.')
```

Program output:

```
Hi,
I am Emil.
```

- **\n** character moves the cursor providing text output to a new line
- printout after **\n** continues in a new line

10.5.3

Tab

- **\t** character is used as a tabulator
- indents following text to the imaginary nearest tab position

```
print('Mother: \tteacher')
print('Father: \tclerk')
print('Daughter: \tpupil')
print('and Son: \tpupil')
```

Program output:

```
Mother:      teacher
Father:      clerk
Daughter:    pupil
and Son:     pupil
```

10.5.4

```
print('my own dog:\tKejsy')
```

```
print('neighbors dog:\tGoulash')
```

Program output:

```
my own dog:  Kejsy
neighbors dog:      Goulash
```

- sometimes it is necessary to use more tabs for correct indentation
- depending on the compiler the tab replaces e.g. maximum 8 spaces (or 4)
- the character after the tab always starts at the position 8*x + 1 (9,17,25 etc.).

After adding the tabs we get to the required outcome.

```
print('my dog:\t\t\tKejsy')
print('neighbors dog:\tGoulash')
```

Program output:

```
my dog:                     Kejsy
neighbors dog:      Goulash
```

10.5.5

Print() function

- special settings allow changing the form of the printout
- **sep** determines which character to insert as a separator in the **print()**; the defualt setting is space
- **end** determines which character is inserted at the end of the written text; by default the character **\n is** set, which is wrap - moving the cursor for the printout to a new line

The simplest use is to replace the separator with a line terminator.

- string separated by commas in the command are written on separate lines:

```
print('one', 'two', 'three', sep = '\n')
```

Program output:

```
one
two
three
```

- if we enter a semicolon as separator

```
print('one', 'two', 'three', sep = ';')
```

Program output:

```
one;two;three
```

- if necessary, the separator can also contain more characters, e.g. '; ' or '...'

10.5.6

- the **end** parameter determines what sign or string is supposed to be printed after the printout of all the text stated in the **print()** function
- by default it is set to **\n** which causes the each printout to move the cursor to a new line at the end
- changing the **end** parameter ensures that the cursor does not move to a new line after each printout

```
print("Hi", end = ", ")
print("long time no see", end = ", ")
print("how are you?")
```

- first two commands print a comma and a space after the text
- the last printout does not have the **end parameter** changed so there will be a delination that would be reflected in the next run

10.5.7

print() with f

- in some printouts, the division of the text into text and variables complicates clarity

```
x = 10
y = 20
z = x + y
print('The sum', x , 'a', y, 'is', z, '.')
```

Program output:

```
The sum 10 a 20 is 30 .
```

A simpler form of notation is available for structured statements:

```
print(f'Sum {x} and {y} is {z}.')
```

Program output:

```
Sum 10 and 20 is 30.
```

- the entry before the text in quotation marks or apostrophes itself contains the letter **f**
- based on this the compiler knows, that the content of the following string needs to be modified
- substitutes the value of the corresponding variables instead of the contents of brackets {}

- every space and every character staded inside the formating string will also be reflected in the printout

10.6 Slices

10.6.1

Slice

- we can get to any character in the string using square brackets

```
ret = 'Sagarmatha'
x = ret[2]
print(x)
```

- we can read several characters from the string using the entry that we refer to as **slice**
- it is created by specifying the selection of characters for the variable name
- the slice borders are to generating a list via **range()**

```
x = ret[beginning : end]
ret = 'Sagarmatha'
x = ret[2:4]
print(x)
```

- it starts at the position **2** and ends one position before the entered end value (i.e. it takes the 3rd character as the last)

10.6.2

What will the following code print?

```
ret = 'Michelangelo'
print(ret[0:len(ret)])  # prints
print(ret[0:6])         # prints
print(ret[7:len(ret)])  # prints
```

10.6.3

Step in a slice

- we can also use step in the slice
- the notation then looks like:

```
x = ret[beginning : end : step]
ret = '123456789'
x = ret[1 : 8 : 3]
print(x)
```

- the character selection starts at the 2. character (position 1 – value **2**), it moves by 3 (value **5**) and again by 3 (value **8**) and ends because it has exceeded the value of the end parameter

10.6.4

What will the following code print?

```
ret = '01030527119'
x = ret[3:9:2]
print(x)
```

10.6.5

- if we omit a parameter in the slice entry, Python will automatically fill it in

```
ret = '0123456789'
print(ret[7:])  # # from the character in position 7 to the
end - 789
print(ret[:6])  # from the beginning to the position 6 -
012345
print(ret[:])   # from the beginning to the end
```

- the version with step indication also works

```
print(ret[::2]) # the result is 02468
```

10.6.6

What will the following code print?

```
ret = 'abcdefghijkl'
x = ret[:2] # variable x contains
print(x)
x = ret[::3] # variable x contains
print(x)
```

10.6.7

Negative indexes

- Python also allows indexing with negative values
- negative indexes start numbering from the last character, which has an index of -1
- it proceeds from the last character to the first so that the second character from the end has the index of -s etc.

```
ret = 'Slovakia'
print(ret[-1]) # prints a
print(ret[-2]) # prints i
print(ret[-3]) # prints k
#...
print(ret[-8]) # prints S
```

10.6.8

- we can use negative values even as part of the slice
- the evaluation principle consist of **replacing a negative value with a real index**
- in case the end value is less than the start evalue, the result is empty

```
ret = '0123456789'
x = ret[-1 : -3] # means that it will start on the last
character i.e. index 9 and ends at the third from the end i.e.
7
print(x)
```

- in another code the characters from the 3rd from the end (**7**) to st from the end (**9**), which will not be included in the printout, i.e.: **78**

```
ret = '0123456789'
x = ret[-3 : -1]
print(x)
```

10.6.9

Negative step

- during the selection of characters, it proceeds by going from a larger index to a smaller one
- therefore the index at the first position must be larger than the index on the second

```
ret = 'Solomon'
x = ret[5:1:-1]
print(x)
```

Program output:

```
omol
```

- we can replace indexes by their negative values

```
ret = 'Solomon'
x = ret[-2:-6:-1]  # it starts at the penultimate and proceeds
to the left
print(x)
```

10.6.10

Omitting values

- if we omit the values in the slice entry and set only the step, in the case of a negative step Python will fill them in by starting with the last and ending with the first character
- prints the text in reverse order – the result starts from the last character

```
ret = 'Solomon'
x = ret[::-1]
print(x)
```

10.7 Basic functions

10.7.1

Removing spaces

- the removal of spaces from the end and beginning of the string provides the **strip()** function
- it is used as part of each string type variable – we separate it from the variable name with a dot
- it does not remove the spaces from inside the text, only from the edges

```
ret = ' Mum has Emma at home.  '
clean = ret.strip() #  contains 'Mum has Emma at home.'
```

10.7.2

Character type

- **ret.isdigit()** tests whether all characters in the string are digits, if so it returns **True**, otherwise it returns **False**
- **ret.isalpha()** tests whether all characters in the string are letters, is so it returns **True**, otherwise it returns false **False**
- **ret.isalnum()** tests whether all characters in the string are letters or digits; if so it returns **True**, otherwise it returns **False** (the string must not contain spaces, commas, brackets etc.)

```
ret = '1.2'
print(ret.isdigit())
ret = 'variable4'
print(ret.isalnum())
```

10.7.3

Letter size

- **ret.islower()** tests whether all characters in the string are from a set of lowercase letters
- **ret.isupper()** tests whether all characters in the string are from a set of uppercase letters
- only letters are checked when executing functions - other characters are not considered

```
ret = 'beta 7'
print(ret.islower())
```

10.7.4

Conversion uppercase -> lowercase and vice versa

- in programs, we usually do not waste time checking whether the string contains only the upper/lowercase letters
- we just convert it to lower/uppercase letters
- **ret.lower()** as a result it returns all letters converted to lowercase letters
- **ret.upper()** as a result it returns all letters converted to uppercase letters
- the original string remains unchanged

```
ret = 'Asta La Vista'
ret1 = ret.lower()
print(ret)   # prints unchanged 'Asta La Vista'
```

```
print(ret1)  # prints changed 'asta la vista'
```

```
ret = 'Asta La Vista'
ret1 = ret.upper()
print(ret)   # prints unchanged 'Asta La Vista'
print(ret1)  # prints changed 'ASTA LA VISTA'
```

While Loop

Chapter 11

11.1 While loop

11.1.1

Loop with an unknown number of repetitions

- sometimes we can not determine the number of repetitions of the loop
- but we know the conditions under which the repetition should continue
- we can provide the execution with the **while** statement and the condition, the fulfillment of which ensures the execution of commands in the body a the loop

```
while condition:
      block of commands
```

- the structure is similar to the **if** function
- the commands contained in the **while** loop are being repeated **until** the condition is met (it is evaluated as **True**)

11.1.2

This is how the loop works:

1. it verifies the validity of the condition
2. if the condition is met, the block of commands is executed
3. the execution returns to the step 1

```
x = 1
while x < 6:
    print(x)
    x += 1
```

- in the block of commands, it is necessary to change the variable which is being tested in the condition
- if we would not do so, the condition would be constantly met and the loop would go on forever
- in our case we had to raise the value x by o 1

11.1.3

Print "Hi" 10 times below.

- the task is the same as when using the **for** loop
- **every task that requires the commands repetitions can be entered trough any type of the loop**

- in this case, however, the programmer has to provide all the operations contained in the **for** structure in separate commands:

- setting the initial value of the control variable
- condition determining the end of the loop
- execution of the commands in the loop
- increasing the value of the control variable

```
i =                 # control variable initialization
while i     :       # while the condition is met, do
  print('Hello')    # command execution
  i =               # increasing the value of the counter
```

11.1.4

Print the even numbers from the interval from 6 to 18 using the while loop.

- we will print the variable content
- the initial value is 6
- it will increase by 2 in each step of the cycle
- the loop will be executed until the value exceeds 18

```
number = 6
while number <= 18:
   print(number)
   number = number + 2
```

We could rewrite the task into a loop with a known number of repetitions as follows:

```
for number in range(6,18,2):
    print(number)
```

11.1.5

- the **while** loop is reffered to as a safe loop
- it tests the fulfillment of the condition first and only then it executes the activity
- if the condition is not already met during the first verification, the commands in the loop **do not even have to run once,** e.g.:

Print all numbers, that lie between two integer limits.

```
lower = 10
upper = 10
i = lower
while i < upper :
   print(i)
```

```
   i = i + 1
print('end')
```

- with the currently set value no number is printed
- the condition is already not met during the first verification - **i** (10) it is not less than **upper** (also 10)
- of course the loop can be executed for different value of the upper and lower limit

11.1.6

Break

- the **break** statement allows a premature termination (interruption) of the loop
- it can be used both in the **while**, and in the **for** loop

```
for i in range(100):
   if i > 20:
       print('it is too much for me')
       break
   else:
       print(i)
print('end')
```

- the loop prints the **i** values and in case it exceeds 20, terminates the loop by the **break** statement - the execution continues with the command after the loop - **print('end')**

The same entry for the **while** loop:

```
i = 0
while i < 100:
   if i > 20:
       print('it is too much for me')
       break
   else:
       print(i)
   i = i + 1
print('end')
```

11.1.7

Complete the program that finds out whether the entered number is a prime number

- a number is prime if between the value 2 and the number one less than our number can not be found any number that would divide it without the remainer
- if we find such number, we do not have to continue with the investigation and we can print that it is not a prime number

```
n = int(input('Enter a number: '))
primenumber = True
i = 2                # investigation starts from value 2
while i              # it will run until i < n
   if n   :          # if n is divisible without a remainder, we
have a divisor
      primenumber = False    # we set that it is not a prime
number
                             # we end the loop execution
   i = i + 1                 # we move to the next value
# here we continue after the end of the loop or the command
if primenumber:
   print('it is a primenumber')
else:
   print('it is not a primenumber')
```

11.1.8

More efficient solution

- we could write the divisibility condition directly into the loop condition
- the loop would ensure increasing of the investigated value until the remainder after the division was zero
- this situation will definitely occur and it will occur at the lates if **i** has the value **n**
- and if **i** has the value **n** after the end of the loop it means that we went trough all smaller values and have not found the divisor

```
n = int(input('Enter a number: '))
i = 2                # investigation starts from value 2
while n % i != 0:    # until I find the divisor
   i = i + 1         # I move to the investigation of the next
number
#---------------------------------
if i == n:           # if I get to the number itslef, it is a
prime number
   print('it is a primenumber')
else:
   print('it is not a primenumber')
```

11.1.9

Check the correctness of the program that detects whether the character 's' is present in the entered text string.

- we can verify the existencion of the character in the condition of the loop and then, when the 's' is found, print that we have found it and end the loop

```
ret = input()
i = 0                       # investigation starts from the
position 0
while ret[i] != 's':        # until I find the character
   # print(ret[i])
   if ret[i] == 's':        # if I found the character
      print('I have it')    # inform the user
      break                 # end the loop
   i = i + 1
```

11.1.10

Correction

- the problem occurs if we try to read a character at the position beyond the end of the string
- such character does not exist and the program would terminate with an error
- the solution is the evaluation of two conditions
- we add the condition that checks whether we are not beyond the last character

```
ret = input()
i = 0
while (i < ?? (ret)) ?? (ret[i] 's'):
   i = i + 1 # moving to the next character in the string

if i   len(ret):
   print('I found')
else:
   print('was not found')
```

11.1.11

Order matters

- if **i** reaches the value **len(ret)** then the attempt to read the character **ret[i]** ends with an error

```
ret = input()
i = 0
while (i < len(ret)) and (ret[i] != 's'):
   i = i + 1
```

- therefore before we enter the condition we verify whether **i < len(ret)**
- if so, **the evaluation of the condition** continues
- otherwise the evaluation of the condition **ends**, because the result (of two conditions that are supposed to apply simultaneously) will be **False,** no matter what the result of the second part of the condition is – therefore, a character outside the string will not be read

- if parts of the condition would be reversed, the program would **crash** every time it went past the last character of the string and tried to compare it to 's'

11.2 Endless loop

11.2.1

Endless loop

- an endless loop is usually undesirable from program execution
- some tasks are easier to be enter using an endless loop, and in the special case, jump out of it using **break** statement

the easiest way to write a still true condition is with **True** (we do not use the "calculation" of the condition, but just enter the result)

```
while True:
    command
```

- the moment we achieve the desired results we can end the loop

```
while True:
    command
    if condition:
        break
```

11.2.2

Input control

- endless loop is often used when testing the value of an input variable
- we want the user to enter a positive integer and we should also check it
- if he did not meet the request we ask for it again and again and again, until the program recieves the required value

- if the user keeps entering a negative value we can continue forever
- after entering the correct value the loop is interrupted and the program continues after the loop

```
while True:
  n = int (input('Enter a positive value: '))
  if n > 0:
    break
print('We can continue')
```

11.2.3

Input control differently

Of course, we can avoid each use of the break statement by adding a suitable condition.

```
n = int (input('Enter a positive value: '))
while n <= 0:
   n = int (input('Enter a positive value again: '))
print('We can continue')
```

Only the programmer's habits and the clarity of the code decide whether to use the version with or without the **break** statement.

11.2.4

List with an unknown number of elements

Write a program that finds a total weight of the passengers on the Titanic. We will not enter the number of passengers at the beginning but we will end the list by entering the value 0.

- since do not know the number of passengers we need to use a loop that will be executed until the value 0 is entered

```
print('Enter the weight of the passengers.')
_sum = 0   # we set the starting value for the sum
while      : # we use an endless loop
 weight = int(input('Enter the weight: '))
 if weight == 0:  # if the termination by 0 was entered
                  # end the loading
 _sum = _sum + weight

print('the total weight is',_sum)
```

Simple Lists

Chapter 12

12.1 Simple lists

12.1.1

Python has its own specifics when working with variables.

One of them is the ability to assign values to multiple variables in one command.

```
a, b, c = 10, 20, 30
print(a,b,c)
```

This entry:

- assigns to the first variable (**a**) the first value entered after the "=" character, i.e. **10**
- to the second variable (**b**) the value **20,** etc.

12.1.2

- it is also possible to set the same value for several variables in one line

```
x = y = z = 20
print(x,y,z)
```

The assignment goes from the right to left:

- first **0** is assigned to the variable **z**
- then the **z** value is assigned to **y**
- eventually the **y** value is assigned to **z**

What is printed after the execution of the following sequence of commands?

```
a = 5
b = 7
c = 10
a, b, c = c, a , b - a
print(a, b, c)
```

12.1.3

Split

- we often need to load more then one value into the program
- the loading of the e.g. three values (names) could look like this:

```
ret1 = input('Enter the 1st name: ')
ret2 = input('Enter the 2nd name: ')
```

```
ret3 = input('Enter the 3rd name: ')
```

- there is also an option to load all three names at once - by entering them in one line
- strings have a split() function
- it can split the text into multiple parts based on the use of the space

E.g.:

```
Ivan Michal Zuzana
```

from the input, **split()** can split into three different values, which are inserted into three different variables

```
ret1, ret2, ret3 = 'Ivan Michal Zuzana'.split()
```

The entire load would look like:

```
ret = input('Enter three names separated by a space: ')
ret1, ret2, ret3 = ret.split()
print(ret1)
print(ret2)
print(ret3)
```

Attention, if the string is not divided into correct number of words, the program will crash.

12.1.4

Split 2

- sometimes the list also contains multi-word values
- e.g., for the first names Milan Rastislav, Adam Ivan etc.
- using the simple split() would be problematic due to the large number of spaces
- **split()** also has a form for such purposes, in which we can enter the dividing character

```
ret = input('Enter 4 names separeted by a comma: ')
a,b,c,d = ret.split(',')
print(a)
print(b)
print(c)
print(d)
```

- input is entered in the form where individual names or double names are separated by commas

- attention, the space after a comma is considered as a normal character in this case

```
Adam,Beata Anna,Jozef František,Ivan
```

12.1.5

Unknown number of elements in list

- the for loop allows iterating trough a list of words as follows

```
for kind in 'cat', 'dog', 'fish', 'hamster':
   print('My favourite animal is', kind)
```

- if we use this cability of the loop and combine it with **split()** function**,** we can read the list from the input separated by spaces or commas with an unknown number of elements in advance
- then we can evaluate each value in the loop individualy
- the following program reads a line of comma-separated values and prints each one of them

```
zoznam = input('Enter words separated by commas: ')
for i in list.split(','):
   print(i)
```

12.1.6

Find the number of time the name entered in the input occurs in the comma-separated list of names.

```
Input: Anna,Beta,Anna,Ivan,Jan,Samuel,Peter,Anna,Jan
Anna
```

```
Output: 3
```

```
# code here
```

Random Numbers

Chapter 13

13.1 Random numbers

13.1.1

Random integer

- useful tool for testing programs, or introducing the element of randomness into a game or programs
- we first need to connect the random number generator to the program - import it
- we get a library (modul) allowing to get random values

```
import random
```

- basic functions:

```
end = 20
rnbr = random.randrange(end) # generates a list using range in
the range 0 to -1 and selects a random number from it
print(rnbr)
```

```
# random.randrange(beggining, end) # generates a random value
from the interval start... end -1
x = random.randrange(-10, 11)
print(x)
```

```
# random.randrange(beginning, end) # selects a random values
from the list range(start, end, step)
x = random.randrange(-10, 11, 2) # generates random even
number in the range -10 to 10.
print(x)
```

13.1.2

Random decimal number

```
import random
x = random.random() # generates a random value from the
interval <0,1)
print(x)
```

```
# random.uniform(beginning, end)  # generates a random value
from the interval <="" pre="">
```

Loop Within a Loop

Chapter 14

14.1 Loop within a loop

14.1.1

Nested loop

- many tasks can be solved using a single loop
- it is not anything unusual to require to use a loop in the body of another loop
- the loop inside another loop is called **nested loop**
- it has the form:

```
for i in range(10):
   for j in range(5):
      command
```

- care must be taken to ensure that the control variables have **different names**

14.1.2

Write a program that prints 1 character 1 in the first line, 2 characters 2 in the second and so on, until 9

- the solution of this task requires two different loops:
- in the first loop we change the digit that is being printed
- in the second we print this digit, the number of printouts is same as the value that is being printed

```
for i in range(10):     # progresses from 1 to 9
  for j in range(i):    # this line ensures the repetition of
the printout i times
     print(i,end="")    # and this printout of the value set in
the first loop without the indentation
  print()               # after printing out i numbers, it
indentates
```

14.1.3

Write a program that draws a square of "o" characters for the entered n.

```
n = int(input('Enter the dimension: '))
for i in range():
  for j in range():
     print('o',='')
```

14.1.4

Write a program that, for the entered integer values m and n, displays m rows below each other and in each there will n circle "o", i.e., an m x n ractangle

```
m = int(input('Enter the number of lines: '))
n = int(input('Enter the number of columns: '))
for i in range(m):
  for j in range(n):
     print('o', end = '')
  print()
```

- the solution is good, but inefficient
- in the nested loop the we always execute the same activity – we always print the character "o" the same number of times
- this operation could be simplified by preparing the entire line (inserting it into a text variable) and then printing it – all of the lines would be printed in one step

```
m = int(input('enter the number of lines: : '))
n = int(input('enter the number of columns: : '))
# we fill the line variable with n characters
line = ''
for i in range(n):
    line = line + 'o'
# we print the entire line m times
for i in range(m):
    print(line)
```

- loops are independent of each other, we may (or may not) use the same control variable
- in the first case we execute the operations in the loop **m** x **n** times
- in the second case we repeat the assignment to the variable **n** times and the and the printout **m** times - the resulting number of operations is **m+n**

14.1.5

Search in string

- **find()** returns the position of the searched substring

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('ga')
print(pos)
```

- if the entered substring does not occur in the string, -1 is returned

- we can use this fact to inform the user or for verification

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('ba')
if pos == -1:
    print('The substring was not found.')
else:
    print('The substring starts at the position', pos, '.')
```

- it is possible to search the string not from the beginning, but from the entered position using a variation of **find()** with two parameters, the second parameter defines the position which the search should start from

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('a',10)
print(pos)
```

14.1.6

Write a program that detects how many time the entered substring is found in it.

```
text = "miss is not accepting kiss as a payment" #input()
substring = "is" # input()
number = 0   # we set the number of found substring
index = 0   # we remember where we started searching, so that
in case of founding substring, we moved past its beginning
# while the substring is located in the string from the
entered position
while text.find(substring, index) > -1:
  # it means the substring was found, increase the number
  number = number + 1
  # and move the index one further than the currently found
substring position
  index = text.find(substring,index) + 1
print(number)
```

Functions

Chapter 15

15.1 Function

15.1.1

Function

- we are familiar with the built-in functions (**random**, **round** a etc.)
- they execute a simple task or return some value
- we can also define our own functions

Custom function:

- solves a certain partial task
- contains the sequence of commands that are repeated several times in a program
- allows the creation of a program hierarchy, increases clarity
- form of the function is:

```
def my_function():
  command1
  command2
  command3
```

- in order for the commands placed in the function to be executed, we need to call for the function at some place in the code

```
...
my_function()
....
```

15.1.2

Types of functions

Function can behave in different ways:

- executing only a sequence of commands (this is called a procedure in some languages)
- executing a sequence of commands and returning a result (this is also called a function in different languages)
- we have already used the returning result functions quite intensively:

```
a = 10.5
b = round(a)
print(b)
```

```
import random
c = random.randrange(10,20)
d = random.randrange(20,30)
print(c, d)
```

- functions mentioned above also have parameters from which the commands inside function are executed

15.1.3

Custom function

- the **def** key word is used for defining the function
- for creating names applies the same rules as for variables

```
def my_function():
  command/s
```

Create a function called greeting, that says Hi.

```
def greeting():
  print('Hi')
```

- however, creating and starting the program in which the function is defined does nothing
- we have to ensure the launch of the function from the main program
- then the entire code file looks like this:

```
def greeting():
  print('Hi')

greeting()
```

- or we can call for the one function several times:

```
def greeting():
  print('Hi')

greeting()
greeting()
greeting()
greeting()
```

15.1.4

Use of functions I.

- the program division into the logical moduls/units
- it allows us to first prepare the whole program and then formulate its individual parts

```
def opening():
  print('Hi I am super program')
  print('I can count to 10')

def core():
  for i in range(1,11):
    print(i)

def end():
  print('... and this is the end')

opening()
core()
end()
```

15.1.5

Functions in different files

- we can use the import statement or the functions that are not built-in inside the Python core

```
import math
```

```
import random
```

- storing custom functions in separate files can be quite useful sometimes

Let's have the file named **functions.py.**

```
# this is a separate file named functions.py
def greeting():
  # write the code of your function here
  print("Hi")
```

- and the file/program that wants to use this function
- we can import this function from the file **functions.py** as follows:

```
from functions import greeting

greeting()
```

- or if we want to use all the function from the file **functions.py** then:

```
import functions

functions.greeting()
```

15.1.6

Function with a parameter

We use the function:

- when we want to logically split the program
- when there is a repeating sequence of commands in the program
- when there is a **similar sequence** of commands differing only in parameter

Write a program that draws rectangles defined by dimensions a and b from the input: a x b, b x a.

- if I had a method available to draw a rectangle with a specified dimensions, I would first call the one rectangle, then the other:

```
a = int(input())
b = int(input())
draw_rectangle(a,b)
draw_rectangle(b,a)
```

- just one more little thing is missing - a function to draw a rectangle
- we need to get its dimensions - parameters in the input
- we can use them as standard variables

```
def draw_rectangle(lines, columns):
  line = 'o' * columns
  for i in range(lines):
    print(line)

# we repeat the body of the program
a = int(input())
b = int(input())
draw_rectangle(a,b)
print()
draw_rectangle(b,a)
```

15.1.7

Parameters

- the function we used had two parameters

```
def draw_rectangle(lines, columns):
  line = 'o' * columns
  for i in range(lines):
    print(line)
```

- parameter is entered as a variable, whose task is to "bring" a value into a function
- by using a variable in a function, we are actually working with the value it represents
- value, that is inserted into the variable, is specified when calling the function

```
draw_rectangle(4, 5)
```

- the order in which the arguments are passed corresponds to the order of the parameters in the function definition
- the first value is inserted into the first variable: **4** to **lines**
- the second one to the variable entered on the second position: **5** to **columns**

When calling the function we can use not only concrete values but also variables

- their values are sent to the function

```
a = 3
b = 7
draw_rectangle(a, b)
```

15.1.8

Parameters and arguments I.

- a function generally looks like:

```
def function_name(parameter1, parameter2 ... parametern):
  command1
  ...
  commandn
```

- parameters are specified in the function definition, in the brackets after its name, separated by commas
- **argument** je a **value**, which enters the parameters of a function

- if parameters re defined for the function, an argument must be entered when calling for it

```
def print_text(text):
  print(text)

print_text()
```

Program output:

```
TypeError
print_text() missing 1 required positional argument: 'text'
```

15.1.9

Parameters and arguments II.

Write a program that draws a rectangle from the "x" characters while the maximum number of characters in the line is entered in the input.

- in this case we make a method, which prints one line and we will call for it from the loop in the body of the program

```
def draw_line(n):
  print('x' * n)

x = int(input())
for i in range(1, x+1):
  draw_line(i)
```

15.2 Global and local variables

15.2.1

Write a program that finds a digit sum of numbers from 100 to 150.

- the solution is simple - for every number we call the function printing its digit sum

```
def digit_sum(number):
  sum_ = 0
  for i in str(number):
    sum_ = sum_ + int(i)
  print(number,'-',sum_)
```

```
for i in range(100,150):
  digit_sum(i)
```

- we used the variable named i in the body of the function and also in the main program - do not they conflict with each other?
- variable used in function (parameters, variables declared in the function body) are **local**
- their existence begins at the point of definition/declaration and ends at the termination of the function
- if the variables in the function have the same name as the variables in the program body, they will overlap it - and will not affect the values of the variables in the body of the program

```
def digit_sum(number):
  sum_ = 0
  for i in str(number):
    sum_ = sum_ + int(i)
  print(number,'-',sum_)

for i in range(100,110):
  digit_sum(i)
print(sum_) # does not exist
```

Program output:

```
100 - 1
101 - 2
102 - 3
103 - 4
104 - 5
105 - 6
106 - 7
107 - 8
108 - 9
109 - 10
NameError
name 'sum_' is not defined
```

- the **sum_** variable does not exist outside the function

15.2.2

Local and global variables

- how does the following code behave?

```
def attempt():
```

```
  a = 10
  b = 20
  print(a + b)

attempt()
print(a)
```

Program output:

```
30
NameError
name 'a' is not defined
```

- variable **a** was declared in the body of the function and is not accessible outside the function - **a** is a local variable

```
def attempt():
  sum_ = 5

sum_ = 10
attempt()
print(sum_)
```

Program output:

```
10
```

- **sum** variable was declared, i.e. was assigned a value in the body of the function, making the function local
- but:

```
def attempt():
  a = sum_
  print('a -',a)

sum_ = 10
attempt()
print('sum_ -',sum_)
```

- **sum** variable in this program is global = declared in the body of the main program
- the variable is not declared in the function, it is just used in it
- if a local variable with the given name does not exist, a global one is searched for
- global variable and its local value is accessible in the entire program, unless it is overlapped by a local variable

15.2.3

Global

- **global** provides a special function - it ensures that the system will treat the variable declared in the function as a global one

```
def attempt():
  global sum_
  sum_ = 5

sum_ = 10
attempt()
print(sum_)
```

- also, if such a variable does not exist at the global level, it will create it as a global one

```
def attempt():
  global sum_
  sum_ = 5

attempt()
print(sum_)
```

15.2.4

Namespaces

- namespaces are the levels, at which identifier (commands or variables) can be created and accessed

There are three levels: built-in(standard), global and local

- built-in namespace (commands and functions of the system core, e.g. int, print...)
- global namespace (names of the global variables, functions, functions imported from moduls etc.)
- local namespace (variables, or other subjects created during the running of the function)

If the system detects the need for the identificator use (variable), it proceeds as follows:

- search in the local tab
- search in the global tab, exist during the running of the program (core)

- search in the built-in namespace (it exist during the function running)

Thanks to this hierarchy it is quite easy to predict the system behaviour.

```
# built-in namespace + variable x
number = 10
my_text = "hello"
x = str(dir())
for i in x.split(', '):
  print(i)
```

15.3 Function with a return value

15.3.1

Functions

- just as values enters a function, we can also retrieves values from it
- obtaining the value as a result of the function's operation is ensured by the **return statement**
- it allows us not to print the result in the function, but to obtain it and use it in another place

Compare:

```
def sum_old(a, b):
  print(a + b)

def sum_new(a,b):
  return a + b

sum_old(20, 30)
x = sum_new(30, 40)
print('the sum is', x)
print('the sum of the numbers 10 and 20 is', sum_new(10, 20))
```

- by obtaining the value the user gets the possibility to do further operations with the result

15.3.2

General notation of the function has the form of:

```
def name(parameter1, ..., paramatern):
  command1
  ...
```

```
  commandn
  return result
```

- while the return statement does not have to be the last command of the function.

Write a function that determines whether an entered number is even or odd.

```
def even(number):
  if number % 2 == 0:
    return True
  else:
    return False

number = int(input())
if even(number):
  print('even')
else:
  print('odd')
```

15.3.3

Write a function that determines whether an entered number is defective (a digit divisible by 3 is contained in the number) and if it is not, the function prints its digit sum.

- this is a contrived example intended to demonstrate the use of the **return** statement

```
def contains(number):
  sum_ = 0
  for i in number:
    digit = int(i)
    if digit % 3 == 0:
      return True
    sum_ = sum_ + digit
  print(sum_)
  return False

number = input()
res = contains(number)
if res:
  print('defective')
```

- the execution of the loop and the entire **contains()** function is interrupted the moment a digit divisible by 3 is found

- the **return** statement has a similar function to a **break** in the loop
- **return** terminates the execution of the function and returns the defined value

15.3.4

Write a function that returns the one of the two entered numbers, whose digit sum is smaller.

- the operation of the digit sum has to be repeated two times
- it would be best to have a separate function that calculates it for us
- in a program, we can use any number of functions that can call for each other

```
def smaller(number1, number2):
  if digit_sum(number1) < digit_sum(number2):
    return number1
  else:
    return number2

def digit_sum(number):
  sum_ = 0
  for i in number:
    sum_ = sum_ + int(i)
  return sum_

print(smaller('1025','5644'))
```

- it is also possible to use the following calling:

```
print(smaller('102',smaller('105',smaller('1025','5644'))))
```

15.3.5

Mechanism of the subprogram execution

When the subprogram calling occurs in the program:

- the return adress is remembered (where it will be necessary to return)
- local variable functions (with an undefined value) and formal parameters with values set according to inputs are created
- program control is transferred to the body of the subprogram
- all the subprogram commands are executed
- local variables are dropped
- control is returned to the place in the program from where the subprogram was called for

15.3.6

Parameter types and result types

- Python does not control data type of the parameters
- the **sum()** function miraculously works both for the text and numeric values

```
def sum_(a, b):
  sum_= a + b
  return sum_

print(sum_(10,15))
print(sum_('10','15'))
```

- however, in case one value is numeric and the other is text, the program crashes

```
print(sum_(10,'15'))
```

- that is why sometimes it is handily to check the data type of the value and react according to it

```
def sum_(a, b):
  if type(a) == type(b):
    return a + b
  else:
    return 'type mismatch'

print(sum_(10,'15'))
```

15.4 Parameter replacement value

15.4.1

Missing parameter

- if parameters are defined for the function, an argument must be specified, otherwise the program returns error **TypeError**.

```
def print_text(text):
  print(text)
print_text()
```

Or:

```
def sum_(a, b):
```

```
  return a * b

print(sum_(4))
```

Program output:

```
TypeError
sum_() missing 1 required positional argument: 'b'
```

15.4.2

Parameter called by name

- standard situation: the number of arguments in a function call is the same as numbers and order of arguments in the function definition
- unusual situation: we want it differently

- the caller identifies the argument by parameter name and the compiler associates the value assignment with the parameter
- it allows to enter arguments in different order

```
def greeting(name, surname):
  print("Hi", name, surname)

greeting(surname="Carrot", name="John")
greeting(surname="Poppy", name="Joseph")
```

15.4.3

The default argument

- the use of parameters called by name does not really solve the situation, when we do not want to enter some of the parameters

```
def greeting(name, surname):
  print("Hi", name, surname)

greeting(surname="Carrot")
```

- in this case we can set some of the parameters in the function header as a default ones
- if the given parameter is not entered, the default one is used
- if we enter the parameter, the entered one is used

```
def print_pupil(name, surname, age = 15):
  if age < 12:
    print(surname, "entered by mistake")
```

```
  else:
    print(name, surname,'is form 8.A and is', age,'years old')

print_pupil('Ivan','Terrible')
print_pupil('John','Small',17)
# print_pupil(name = 'Jana', surname = 'Blue',17)
print_pupil(name = 'Dana', surname = 'Green')
```

Program output:

```
SyntaxError
positional argument follows keyword argument (, line 10)
```

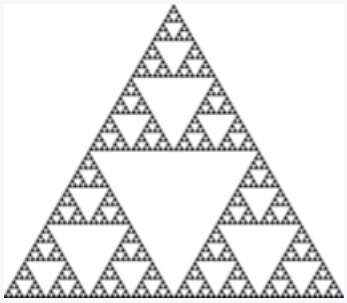
- the default parameters can be omitted when calling the function(but they should take the last places in the list)
- if we name the parameters, all but those that have a defined default value must be specified when calling the function

15.5 Recursion

15.5.1

Recursion

- a common, but often very well-hidden context
- object is recursive, if it is partially composed or defined using itself
- definition of the object using the object itself
- recursive expressions are generally shorter and easier to understand than other expressions, because they describe characteristic properties of a function



in mathematics:

- expressing the following value by the previous one
- $a_n = a_{n-1}+d$

in programming:

- calling a subprogram by the subprogram itself

15.5.2

Recursion in programming

Standard function call mechanism:

- the activation record is stored in the system stack, it consists of:
- copy of method parameters
- return address (the location from which the function was called)
- the execution of the body of the function
- entry into the function
- return to the address stored in the stack
- release of the activation record
- program continues...

Recursive call mechanism - analogy:

- each time the method is called, new copies of the variables/parametersare created and the return address is remembered
- => for each method call, an activation record is stored in the memory stack and the method is executed from the beginning
- after the method ends, the memory is freed and thanks to the informations from the stack, the code continues from the place it was called from
- => each recursion run is memory intensive
- it is possible to get to the potential stack overflow

```
def return_element(n):
  if n == 1:
    return 10
  else:
    return return_element(n-1) + 4

print(return_element(5))
```

15.5.3

The finitude of recursion

- recursions on the opening frames are infinite
- however, practice requires termination after reaching a certain number of repetitions or fulfilling the condition
- situation, in which the method terminates (without calling itself again), is called a trivial case
- when using recursion in programming has to be defined otherwise a stack overflow will occur
- recursion is therefore usually defined in such a way, that the difficulty of the solved task gradually decreases until it reached an elementary result

```
def return_element(n):
  if n == 1:
    return 10
  else:
    return return_element(n-1) + 4

print(return_element(5))
```

15.5.4

Write a program that calculates n! for an entered n.

- **n! = n*(n-1)*(n-2)*(n-3)*...*3*2*1**

- e.g. 5! = 5*4*3*2*1

```
def factorial(n):
  if (n == 0):      # trivial case, ...
    return 1             # ... we have the result, the
function ends
  else:                # recursive case: to the result
    return n*factorial(n-1);    # n* calculation is assigned
for the lower value of the factorial

print(factorial(5))
```

```
def factorial(n):
  print(' '*(10-n), 'I enter the methos with the value',n)
  if (n == 0):      # trivial case, ...
    print(' '*(10-n), 'I am returning the value 1')
    return 1             # ... we have result, function ends
  else:                # recursive case: to the result
    print(' '*(10-n), ' I call the calculation f = ',n,' *
','factorial(',n-1,')', sep='')
    f = n*factorial(n-1)
    print(' '*(10-n), 'I am returning the value',f)
    return f  # n* calculation is assigned for the lower value
of the factorial

print(factorial(5))
```

Program output:

```
      I enter the methos with the value 5
      I call the calculation f = 5 * factorial(4)
       I enter the methos with the value 4
       I call the calculation f = 4 * factorial(3)
        I enter the methos with the value 3
        I call the calculation f = 3 * factorial(2)
         I enter the methos with the value 2
         I call the calculation f = 2 * factorial(1)
          I enter the methos with the value 1
          I call the calculation f = 1 * factorial(0)
           I enter the methos with the value 0
           I am returning the value 1
          I am returning the value 1
         I am returning the value 2
        I am returning the value 6
       I am returning the value 24
      I am returning the value 120
```

```
120
```

15.5.5

Write a recursive method that returns the power of the entered number.

- **power a^n**
- **e.g. $5^4 = 5 * 5 * 5 * 5$**

- recursively, the power can be expressed as $a^n = a*a^{n-1}$
- e.g. $5^4 = 5*5^{4-1=3}$
- the trivial case occurs if the exponent is 0 (or 1)
- each previous one is non-trivial
- the exponent increases by one at each subsequent level

```
def power(a, n):
   if n == 0:       # trivial case,
     return 1;
   else:                  # the result is a * power a, but a
degree lower
     return a * power(a,n-1);

print(power(2,5))
```

```
def power(a, n):
  print(' '*(10-n), 'I am entering the method with a
value',a,n)
  if (n == 0):       # trivial case, ...
    print(' '*(10-n), 'I am returning the value 1')
    return 1              # ... we have result, the function
ends
  else:                 # recursive case: to the result
    print(' '*(10-n), ' I call the calculation m = ',a,' *
','power(',a,',',n-1,')', sep='')
    m = a*power(a, n-1)
    print(' '*(10-n), 'I am returning the value',m)
    return m  # n* calculation is assigned for the lower value
of the calculation

print(power(3,4))
```

Program output:

```
        I am entering the method with a value 3 4
        I call the calculation m = 3 * power(3,3)
         I am entering the method with a value 3 3
         I call the calculation m = 3 * power(3,2)
```

```
          I am entering the method with a value 3 2
          I call the calculation m = 3 * power(3,1)
           I am entering the method with a value 3 1
           I call the calculation m = 3 * power(3,0)
            I am entering the method with a value 3 0
            I am returning the value 1
           I am returning the value 3
          I am returning the value 9
         I am returning the value 27
        I am returning the value 81
81
```

Exceptions

Chapter 16

16.1 Exceptions

16.1.1

Exception

- exception, or an exceptional state is a situation in the program that occurs in the event of an error during program execution and usually interrupts/terminates program execution
- we can predict many error and treat them directly with the program

```
a = 1
b = 0
if b == 0:
  print("the quotient is not");
else:
  print("quotient:", a / b);
```

- in the code, we actually find out whether the variable **b** does not contain a value that could cause the program crash, and if so, we do not allow the dangerous operation

16.1.2

- however there are situations in which programming a predictable treatment is difficult
- the typical example is trying to convert a read value to a number

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

Program output:

```
 testValueError
invalid literal for int() with base 10: 'test'
```

16.1.3

Try-except

- through a pair of commands, we can create blocks in the code that allow us to catch the exception and deal with it

```
# some opening code
try:
  # commands in which an error could occur
```

```
except:
  # what to do if an error occurred
# code extension
```

- then the previous problem can be solved as follows:

```
ta = input()
try:
  a = int(ta)
  print(a)
except:
  print('an error occurred after the conversion')
print('the code continues')
```

Program output:

```
 demoan error occurred after the conversion
the code continues
```

16.1.4

Range of try

- it is up to debate, how big the try block should be
- program from the previous task:

```
ta = input()
try:
  a = int(ta)
  print(a)
except:
  print('an error occurred after the casting')
print('code continues')
```

- can be also adjusted to:

```
ta = input()
try:
  g = int(ta)
except:
  print('an error occurred after the casting')
print(g)
print('the code continues')
```

Program output:

```
 twoan error occured after the casting
NameError
```

```
name 'g' is not defined
```

- however, in such case there is a risk that the variable **g** will be undefined
- the correct solution would be:

```
ta = input()
g = -1
try:
  g = int(ta)
except:
  print('an error occurred after the casting')
print('the code continues')
print(g)
```

- or:

```
ta = input()
try:
  h = int(ta)
except:
  print('an error occurred after the casting')
  h = -1
print('the code continues')
print(h)
```

16.2 Types of exceptions

16.2.1

Various types of errors

- there are situations in which a number of errors may occur on several lines
- we need to inform the user what kind of error occurred and execute a different code for each error
- for this purpose, we use **except**, for which we indicate the error for which its code is intended
- of course we need to be familiar with the errors that can occur

```
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-numeric value was entered')
```

- the **ZeroDivisionError** exception occurs in case of division by zero
- the **ValueError** exception occurs in case the input value can not be cast to an integer of the **int** type

16.2.2

The most common types of integers

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

16.2.3

General exception

- our goal is to handle expected exceptions
- also to ensure that the program completes its activity even in case of an exception that was not identified by us

We adjust the program from the previous task:

```
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-numeric value was entered')
```

- if in this case an exception of a different type than those specified in the list occurs, it will not be processed and the program will end
- therefore we will complete the block intended for processing all other exceptions

```
try:
  a = int(input())
  b = int(input())
  print(a // b, a % b)
  print(a[2])
  print('done')
except ZeroDivisionError:
  print('we do not divide by zero')
except ValueError:
  print('a non-numeric value was entered')
except:
  print('something else')
```

Program output:

```
 2 12 0
something else
```

16.2.4

Exception details

- even if we use a general exception we will not lose detail information about it
- every exception that occurs fills a variable with information about the situation, which we can access through the following notation:

```
try:
  # some code causing the exception
except Exception as err:
  print(err)  # print information about the error
```

- usage example:

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

Program output:

```
 10 0error: integer division or modulo by zero
type:
```

```
details: ('integer division or modulo by zero',)
```

- the **Exception** type used represents a general exception
- all other exceptions are derived from it

16.2.5

Else as a part of try

- in case no exception occurs, we can ensure the execution of commands as follows:

```
try:
    # commands
except Exception1:
    print('the input must contain integers')
except Exception2:
    print('division by zero')
else:
    print('everything is okay')
```

16.2.6

Finally

- the last part of the **try** block is the **finally** part
- it is intended for commands to be executed regardless of whether or not an exception occurred in the **try** block

```
try:
    # commands
except Exception1:
    print('the input must contain integers')
except Exception2:
    print('division by zero')
else:
    print('everything is ok')
finally:
    print('and this will be executed everytime')     # the
command after the try block has the same effect :)
```

16.3 Exception generation

16.3.1

Exceptions inside functions

- there are situations when we not only process the exception, we also have to generate it = force it
- this most often happens inside functions
- different functions handle the different exceptions differently

E.g.:

```
# by returning the value -1
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
```

16.3.2

Raise

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

- in this case returning the value -1 or 0 would be inappropriate
- the best procedure is to generate an exception and, unless it is handled in the body of the program, interrupt its execution
- the **raise** command is used to create an exception
- it is used in combination with an existing type of error

```
def truncate(numerator, denominator):
  if denominator == 0:
```

```
    raise ValueError('Denominator is zero')
  a, b = numerator, 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
```

- in case of error this version of the code will not finish
- therefore, we will use the **try - except** block to catch and process the exception

```
def truncate(numerator, denominator):
  if denominator == 0:
    raise ValueError('Denominator is zero')
  a, b = numerator, 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)

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

Program output:

```
An error occurred: Denominator is zero
end
```

Files

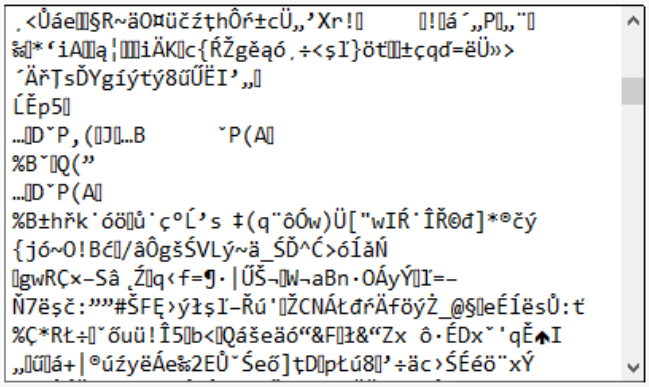
Chapter 17

17.1 Text file

17.1.1

Files

- they serve for storing data
- first form - **binary files** into which data was entered so that it could be processed as quickly as possible
- entered characters represented coded numbers or text in the form in which they were stored in the computer's memory



Advantages:

- fast prcessing

Disadvantages:

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

Newer form - **text files:**

- the standard for data storage became text files (more precisely, files with text understandable to the user)
- thanks to the fast hardware, the time aspect is not a problem

```
    <v-content>
    <app-bread v-if="isLogged" />
    <v-alert v-if="getFeedback" color="red">
{{getFeedback}}</v-alert>
      <router-view></router-view>
    </v-content>
  </v-app>
</template>

<script>
  import Header from './components/header/header.vue'
  import Bread from './components/elements/Bread.vue'
  import {mapGetters} from 'vuex'
```

- the data are 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
- working with text files is usually not done by characters, but by lines

17.1.2

Text file

- sequence of character
- lines end with the **\n** character

```
Physics, mathematics, IT sector, technology. In these and many
other fields, women get the short end of the stick
professionally.
Succeeding in the world of science and research, which has
been the domain of men for many years, is very challenging for
women. They encounter prejudice or discrimination and do not
always have suitable conditions to combine work with family
life.
The solution, although it is a long shot, but it exists: stop
putting small children in boxes and do not prevent girls from
"boyish" games and hobbies.

According to data from the Slovak Commission for UNESCO, far
fewer women than men work in the scientific sector – only 33%
worldwide.
```

17.1.3

File operations

- creation
- data notation (sometimes addition)
- reading data
- for these operation we need: opening and closing the file

17.1.4

File creation

- a special command for the file creation does not exist
- we always only open the file - for reading or entering
- in the case of opening for entering, the file will be created if it does not exist
- the command for opening has the form :

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

Program output:

```

```

- the variable **f** ensures the connection with a file - we will use it to refer to the relevant commands
- the **open** function ensuures the creation of this connection
- the connection will be directed to the '**data.txt**' file located in the same folder as the program
- the second parameter of the open function determines whether we open the file for writing (w) or reading (r)

Currently we can enter the data into the file, with two basic approaches existing:

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

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

- using the **write** function which is part of the file variable
- in this case, we also need to enter **\n** at the end of the line for delineation

```
f.write('hello second\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

17.1.5

Loading data from a file

- we usually save data to a file so that we can read it
- opening a file is analogous to opening for writing

```
# write
f = open('data.txt','w')
print('hello', file = f)
f.write('hello second time\n')
```

```
f.write('the end')
f.close()

# open
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 second time

the end
```

- the loaded line from the file already contains the end of the line
- we will therefore adjust the print so that it does not cancel

```
# open
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 second time
the end
```

17.1.6

Loading an unknown number of lines

- by sequentially executing the **readline()** methods, we load each line, but if we do not know the exact number of them, we will either cause an error or not read the entire file
- it is much more reasonable to use a **while** statement that ends when an empty line is read = end of file

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

- we can also use a **for** loop just as well:

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

17.1.7

Exception handling

- so far we have ignored the possibility of an exception, but it is dangerous

When **entering data,** we can:

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

When **opening** a file:

- again it may be the wrong way to the file
- we do not have read permission
- access to the file is lost etc.

```
# entry
try:
  f = open('data.txt','r')
  print('hello', file = f)
```

```
  f.close()
except Exception as err:
  print('error:',err)
```

Program output:

```
error: not writable
```

```
# opening
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'
```

17.1.8

Handling simplification

- when working with files it is recommended to use the **with** command structure:

```
try:
  with open('data22.txt','r') as f:
    for line in f:
        print(line, end='')
except Exception as err:
  print('error:',err)
```

- after the commands in the **with** block are executed, the open **file is automatically closed**

17.1.9

Loading the entire file

- in some situations we do not need to load row by row
- we need the content of the file as a continuous string

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

```
text = f.read()
print(text)
f.close()
print('length:', len(text))
```

Program output:

```
hello
hello second time
the end
length: 31
```

17.1.10

Printout with hidden characters

- if we take the program...

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

- ... the number of characters in the output and in the string does not quite match
- if we want to see also hidden characters (e.g. \n) we use the function **repr()**

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

Program output:

```
'hello\nhello second time\nthe end'
lenght: 31
```

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

17.1.11

Adding data to the end of the file

- sometimes there is a need to add text to the end of the file
- it is a combination of opening for entry and setting to end of content
- parameters 'a' will execute this operation automatically when opening the file

```
# file preparation
with open('data.txt','w') as f:
  print('beginning', 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)

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

17.2 Typical tasks

17.2.1

Find the number of the words if the file.

- words are separated by spaces
- the problem is, we do not know whether there is a space at the end of the line
- we need to read line by line and strip each line of spaces
- subsequently, through split, we find out the number of elements in the created list

```
# text preparation
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)
```

```
number = 0
try:
  with open('long_text.txt','r') as f:
    for line in f:
      cleaned = line.strip()
      number = number + len(cleaned.split(' '))
      print(number)
except Exception as err:
  print('error:',err)
```

Program output:

```
5
65
152
193
```

17.2.2

Copy the content of the files.

- a typical task on language functionality
- we can copy files at once
- or by lines and possibly do some operation with the lines

```
# text preparation
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)

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

- by lines:

```
try:
  with open('long_text.txt','r') as fr, open('copy2.txt','w')
as fw:
    for line in fr:
      fw.write(line)
except Exception as err:
  print('error:', err)
```

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

Program output:

```
Testing line 1 2 3
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.
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 .
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.

end
```

17.2.3

Find out the student's average from subjects that are stored together with grades in a file, separated by commas (mathematics, 1, 2, 3) and print this average in the second file.

- we load an unknown number of data in a line and split it
- we have to deal with the first data that is not a number - we remember it as an object

```
# data preparation
try:
  with open('grades.txt','w') as f:
    print('english,1,2,1,1,1,3',file=f)
    print('mathematics,1,2,1,5,3',file=f)
    print('chemistry,4',file=f)
    print('biology,1,5,3',file=f)
except Exception as err:
  print('error:', err)
```

```
# data preparation
try:
  with open('grades.txt','r') as fr,
open('certificate.txt','w') as fw:
    for line in fr:
      subject = ''
      sum_ = 0
      number = 0
      grades = line.strip().split(',')  # without strip
      for z in grades:
        if z.isdigit():
          sum_ = sum_ + int(z)
          number = number + 1
        else:
          subject = z
      print(subject +','+str(sum_/number), file=fw)
except Exception as err:
  print('error:', err)

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

Program output:

```
english,1.5
mathematics,2.4
chemistry,4.0
biology,3.0

end
```

Lists

Chapter 18

18.1 List

18.1.1

List

- more than 90% of applications do not work with simple data, but with lists
- typical lists: persons, invoices, web addresses, measured values, etc.
- required operations: adding and deleting data, various calculations, arranging, etc.
- the simplest list: a string of characters

18.1.2

List creation

- most simply by listing the elements
- the list is delimited by curly (square) brackets, with elements 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 Shed']
years = [1945, 1969, 1971, 1978, 1980, 1984, 2012, 2015, 2019]
```

- Python also allows storing elements of different types in a list

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

- we can also create an empty list:

```
a = []
```

- or:

```
a = list()
```

- if we let the type from the list be printed, we get a **list**

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

Program output:

```
a is
```

```
b is
```

18.1.3

List output

- by default through the **print** command
- the result is enclosed in braces

```
temperatures = [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
students = ['Joseph', 'Ivan', 'Michal', 'Anna']
print(temperatures)
print(students)
```

Program output:

```
[36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
['Joseph', 'Ivan', 'Michal', 'Anna']
```

18.1.4

Access to the list elements

- provided through an index
- rules same as for indexes in string

```
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
```

18.1.5

Data editing

- elements in a list can be changed
- entry

```
list_[i] = value
overwrites the content of the i-th element
```

E.g.:

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

18.1.6

The number of elements in the list

- **len** - function returning the number of elements

```
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))
```

18.1.7

List loading

- the simplest and already used is the use of **split()**
- the result of the split function is a list - **list**

```
data = input('Enter sa list separated by commas').split(',')
print(data)
```

Program output:

```
Enter sa list separated by commas 1,2,3,4,5,6['1', '2', '3',
'4', '5', '6']
```

18.1.8

Adding elements to the list

- addition is provided by the **append()** command

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

Program output:

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

- this is probably one of the first uses of a command that is part of some data structure
- the **list** structure is programmed in such a way that it makes the functions that are part of it and manipulate the data stored in the given structure available to the programmer
- they are separated from the specific variable representing the list by a dot
- these functions are referred to as **methods**

18.1.9

Alternative addition

- we can also add an element to the end of the list by using [] to create a one-element list from it
- and add it to the existing list using the '+' operation

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

Program output:

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

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

Program output:

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

18.1.10

Sequential loading of list elements

- elements can be added to the list gradually
- we always read one and add it to the list - usually at the end
- we start the solution by creating an empty list
- after entering the terminating element, e.g. '0', loading stops

```
list_ = []
while True:
  element = input()
```

```
  if element == '0':
    break
  list_.append(element)
print(list_)
```

18.1.11

Element location

- if we want to place the element in a specific position, and not at the end, 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

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

Program output:

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

18.1.12

Divide the loaded elements by putting negative values on the left side of the list and positive values on the right side.

- according to the value of the element, we decide whether to insert it at the beginning using **insert()** or at the end using **append()**
- since we're only working with text so far, not numbers, we test if the first character is '-'

```
list_ = []
while True:
  element = input()
  if element == '0':
    break
  if element[0] == '-':
    list_.insert(0, element)
  else:
    list_.append(element)
print(list_)
```

18.2 Working with data in a list

18.2.1

Traversing the list

- the operations that are executed on an array usually require processing each element
- we implement the transition through a cycle from the first element to the last one located at the position len(list) - 1

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

- or by just iterating over the elements of the list

```
list_ = ['Adam', 'Berta', 'Cecil', 'Dana', 'Ema', 'Fero',
'Gusto', 'Hana']
for item in list_:
  print(item)
```

18.2.2

Enumerate

- **enumerate()** is a special function that, in addition to the list, also ensures its indexing
- creates a pair of index and element from the list, which we can use in the corresponding cycle

```
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
```

- it also allows you to set the beginning of index numbering:

```
people = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
```

```
for index, person in enumerate(person, start = 1):
   print(index, person)
```

18.2.3

Searching for a value in a list

- the easiest way to formulate a question about the existence of an element in the list is by using the **in** keyword

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

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

ex = 'Joseph'
if ex in list_:
  print(ex,'is there')
else:
  print(ex,'is not there')
```

Program output:

```
Juraj is there
Joseph is not there
```

- if we also want to find out what position it is in, we can use the **index()** method:

```
list_ = ['Peter', 'Pavol', 'Michael', 'Juraj', 'Jan']
boyfriend = 'Juraj'
print(list_.index(friend))
```

- if the element is not in the list, it generates an exception:

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

- index can also have two other parameters:

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

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

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

value = 4
try:
  print(list_.index(value))
except:
  print(value, 'not found')

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

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

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

Program output:

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

18.2.4

Deleting an element

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

- using the **del()** statement, which deletes the element at the entered position

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

```
print(list_)
```

- using the **remove()** method
- which deletes the first occurrence of an element with the entered content
- when it has nothing to delete, it will generate an exception

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

18.2.5

Other operations over the list

- **pop()** - deletes the last element from the list and returns it as the result of the call

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

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

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

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

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

18.2.6

Numerical list

- in many tasks we need to convert the list obtained from the input to an integer or decimal number
- we usually create a new list
- for data read from input and separated by commas, the code might look like this:

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

- or:

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

- if we also assume an empty input, then we have to process it after reading the input
- the result of the **split()** method is always a list, so in the condition we check if it is not empty

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

18.2.7

Operations over (not only) a numeric list

- Python provides several functions that work with a list of list elements
- they have defined different behavior for different data types of 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

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

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

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

18.2.8

Lists joining

- it is possible to connect lists using '+'
- connection 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)
```

18.3 Iterations and slices

18.3.1

List creation

- by entering an empty list

```
a = []
```

- using the **list()** list

```
a = list()
```

- using the **list()** function with a parameter that will provide an iterable element, or list

For example:

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

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

```
a = ['mathematics','IT', 'physics','chemistry']
b = list(a)
print(b)
```

Program output:

```
['mathematics', 'IT', 'physics', 'chemistry']
```

- in the last example, a **copy** of the original list is created in list **b**, which is independent of list **a**

18.3.2

Creating a list from a file

- when opening a file, the input is a list of lines, similar to using **for**
- the **open()** command returns a list of lines, which can be enumerated via:

```
list(open('data.txt', 'r'))

# 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)
```

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

18.3.3

Non-iterable objects

- some objects cannot be used to create a list

```
list_ = list(10)
```

Program output:

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

```
list_ = list(True)
```

Program output:

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

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

Program output:

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

```
list_ = list('m')
# it is ok
```

18.3.4

Slices

- as with strings, lists also support slices
- the syntax is exactly the same, the result is a new list

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

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

Program output:

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

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

Program output:

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

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

Program output:

```
[]
```

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

Program output:

```
[5, 6]
```

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

Program output:

```
[7, 6]
```

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

Program output:

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

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

Program output:

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

18.3.5

Iterator betrayal

How are the following programs different?

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

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

- in the first case, we only change the value of the variable, which changes in each step
- the value of the i-th element is inserted into it
- it cannot change the content of the variable whose value it has acquired
- in the second case, we directly change the content of the i-th element of the list

18.4 List comprehension

18.4.1

List comprehension

- a special feature of Python language that makes it a brutally powerful language
- is used to generate new lists from the existing ones
- it can be thought of as a filter
- in square brackets we indicate the expression on the basis of which the new element is formed
- the creation of a new element is repeated according to the elements iterated in the **for** section
- and this can be supplemented by a condition

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

- creating a list based on the elements defined by the cycle

```
# insert i in the list, which you get as an element of the
cycle iteration for range(10)
list_ = [i for i in range(10)]
print(list_)
```

Program output:

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

```
# insert triple i into the list, which you get as an element
of the cycle iteration for range(10)
list_ = [i*3 for i in range(10)]
print(list_)
```

Program output:

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

18.4.2

Go through the list

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

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

Program output:

[2, 10, 16, 14, 4]

18.4.3

Working with text

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

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

Program output:

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

18.4.4

Filter

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

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

Program output:

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

18.4.5

Filter text

- **From the existing list of names, add those that do not contain 'a' to the new list.**

```
list_ = ['anna', 'beata', 'cynthi', 'daniel', 'kitti']
new = [name for name in list_ if name.find('a') == -1]
print(new)
```

Program output:

```
['cynthi', 'kitti']
```

Lists and Memory

Chapter 19

19.1 List manipulation

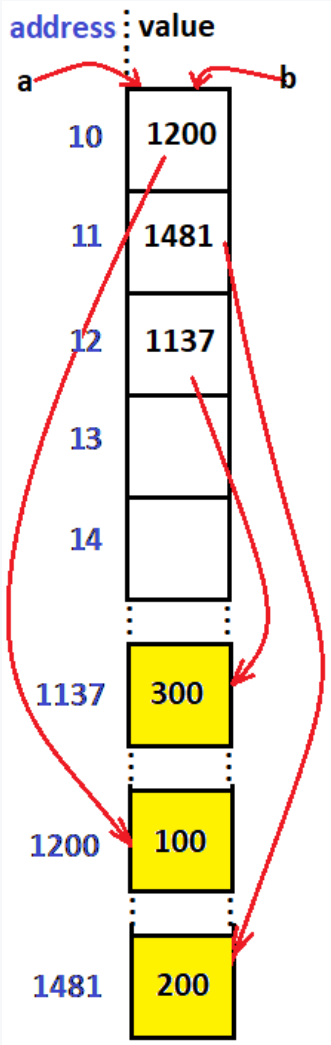
19.1.1

List in memory

- we can create a list in memory, e.g. by listing the elements

```
a = [100, 200, 300, 400, 500]
```

- the necessary memory space is allocated for it - in this case 5 spaces
- element values are not stored in the list
- each element of the list is stored in "random" locations in memory, and the list only contains the **addresses** of those random locations



- if we make an assignment

```
b = a
```

- it means that the same list in memory is referenced by two variables (as in the picture)
- rewriting an element at a specific location in the list rewrites that element for both lists

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

Program output:

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

- such behavior is typical of lists or more complex data structures
- in the case of simple variables of standard types, assigning values between variables just copies them

19.1.2

Adding an element

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

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

```
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']
```

- a list like a list of addresses in memory is created in memory **in a contiguous block** for fast access to elements
- using the **append()** command will keep the contiguous block
- if necessary, creates a new contiguous block and copies the original elements into it
- the minimum number of elements in a block is around 200 on the first use

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

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

Program output:

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

- the assignment in line 4 creates a new variable that it gets
- as the contents of the original list and the list with one element - that is, the allocation of new space will take place
- this breaks the binding of a and b

19.1.3

Behavior in basic operations

- slice returns a list of elements by default, so it does not change the original list

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

- similarly comprehension

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

19.1.4

Assignment to slice

- in addition to retrieving a slice from a list, we can change the list by assigning a list of elements to the slice, e.g.:

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

Program output:

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

- in this case, two elements were replaced by two new elements
- it is also possible to implement an operation that replaces some number of elements with another number of elements
- however, such operations are supported specifically in the Python language - in "classic" programming languages such an operation is meaningless

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

Program output:

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

- or four elements are replaced by three

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

Program output:

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

What is the result of the following assignment?

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

Program output:

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

19.1.5

Elements comparison

- the comparison is the same as for strings
- it is traversed through the elements of the list, and in the case of the first difference, the elements are compared

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

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

```
print(a < b)
```

- strings:

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

Program output:

```
False
```

- the combination of text and numeric value is problematic

```
a = ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear']
b = ['alphabet', 1, 2, 3, 5, 4]
print(a < b)
```

Program output:

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

19.1.6

Ordering list elements

- we have the following functions:
- **sorted()** - as a result of the operation it returns the ordered elements of the list, the original list is not changed

```
a = ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear']
print('a before:',a)
print('a sorted:', sorted(a))
print('and after:', a)
```

Program output:

```
a before: ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear']
a sorted: ['alphabet', 'ate', 'bear', 'grandpa', 'on', 'told']
and after: ['alphabet', 'ate', 'grandpa', 'told', 'on',
'bear']
```

- **sort()** - list method - rearranges the elements => the list changes

```
a = ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear']
print('a before:',a)
a.sort()
print('and after:', a)
```

Program output:

```
a before: ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear']
and after: ['alphabet', 'ate', 'bear', 'grandpa', 'on',
'told']
```

- operations between incompatible types are still not allowed

```
a = ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear', 4]
print('a before:',a)
a.sort()
print('and after:', a)
```

Program output:

```
a before: ['alphabet', 'ate', 'grandpa', 'told', 'on', 'bear',
4]
TypeError
'<' not supported between instances of 'int' and 'str'
```

19.1.7

The input contains a list of names separated by commas. Find out which name appears the most times in the list.

```
Adam, Dana, Jana, Adam, Jana, Zuzana, Adam, Jana, Jana

data = input().split(', ')
data.sort()
maxname = data[0]
maxnumber = 0
number = 0
name = data[0]
print(data)
i = 0
while i < len(data):
  while i < len(data) and name == data[i]:
    i = i + 1
    number = number + 1
  if number > maxnumber:
    maxnumber = number
    maxname = name
  if (i == len(data)):
    break
  name = data[i]
  number = 0
```

```
print(maxname, maxnumber)
```

19.1.8

Layout parameters

- the **sorted()** function has two optional parameters - **reverse** and **key**
- **reverse** allows you to reverse the sort order

```
list_ = [1, 2, 8, 4, 5]
z2 = sorted(list_, reverse = True)
print(z2)
```

Program output:

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

- key allows you to define rules for ordering through a function
- e.g. to order the values based on the absolute value, we use:

```
list_ = [-1, 2, 8, 4, -5]
z2 = sorted(list_, key = abs)
print(z2)
```

Program output:

```
[-1, 2, 4, -5, 8]
```

- it is common to arrange strings regardless of character size using the **lower** function from the **str** package, e.g.:

```
a = ['alphabet', 'Ate', 'grandpa', 'Told', 'on', 'bear']
print('a sorted:', sorted(a))
print('a sorted lower:', sorted(a, key = str.lower))
```

Program output:

```
a sorted: ['Ate', 'Told', 'alphabet', 'bear', 'grandpa', 'on']
a sorted lower: ['alphabet', 'Ate', 'bear', 'grandpa', 'on',
'Told']
```

- as a **key** parameter, we actually enter the name of the function, which should modify the input before the actual comparison

19.1.9

Join

- the function that allows us to process the list elements given as a text string is **split()**
- its opposite is the **join()** function, which creates a text string from the **list** with a separator that we enter
- **join**() is a text string method of the form:

```
list_ = list()
separator = ' '
result = separator.join(list_)
```

- e.g. 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:

```
 mother has small child at home['mother', 'has', 'small',
'child', 'at', 'home']
mother;has;small;child;at;home
mother--has--small--child--at--home
```

19.2 List as a parameter

19.2.1

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 sum_(a, b):
```

```
  return a + b

s = sum_(10, 20)
print(s)
x = 10
y = 5
z = sum_(x, y)
print(z)
```

Program output:

```
30
15
```

19.2.2

List as parameter I.

- list is the standard input to the function
- we can scroll through it or evaluate individual elements
- the sum of the elements in the list:

```
def my_sum(z):
  s = sum(z)
  return s

list_ = [1, 2, 3, 4, 5]
z = my_sum(list_)
print(z)
```

Program output:

```
15
```

- number of occurrences of an element:

```
def count(element, list_):
  poc = 0
  for i in list_:
    if i == element:
      poc = poc + 1
  return poc

list_ = [1, 2, 3, 2, 5, 2, 3, 1]
z = count(2, list_)
print(z)
```

Program output:

```
3
```

19.2.3

List as a parameter II.

- in the case of a variable representing a list, its content is not independent of the original variable
- standard function:

```
def my_sum(a, b):
  a = a + 5
  return a + b

x = 10
y = 5
print('original:', x)
z = my_sum(x, y)
print('new:', x)
```

Program output:

```
original: 10
new: 10
```

- function with a list:

```
def my_sum(z):
  x = sum(z)
  z.append(x) # we will also add sum_ to the end of the list
  return x

list_ = [1, 2, 3, 4, 5]
print('original:', list_)
z = my_sum(list_)
print('new:', 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
- reason - slowness of copying list contents, copying properties of other 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 exiting the function
- this behavior can be used very effectively

19.2.4

Adjust the list of elements by zeroing negative values and doubling positive ones. Execute the operation in the function.

```
def operation(z):
  for i in range(len(z)):
    if z[i] < 0:
      z[i] = 0
    else:
      z[i] = z[i] * 2

list_ = [-1, 2, -3, 4, 5]
operation(list_)
print(list_)
```

Program output:

```
[0, 4, 0, 8, 10]
```

- we implemented the content change by rewriting the list elements => the original list was changed
- of course, the task can be solved without changing the original list

```
def operation(z):
  n = list()
  for i in range(len(z)):
    if z[i] < 0:
      n.append(0)
    else:
      n.append(z[i] * 2)
  return n

list_ = [-1, 2, -3, 4, 5]
new = operation(list_)
print(list_)
print(new)
```

Program output:

```
[-1, 2, -3, 4, 5]
[0, 4, 0, 8, 10]
```

19.2.5

Creating a list

- we often need to create an empty list filled with some initial value
- thanks to the direct access to the list, we can provide this directly in the universal method

```
def create_list(number, default = 0):
    return [default] * number

z1 = create_list(5, 1)
print(z1)
z2 = create_list(5)
print(z2)
z3 = create_list(4, '')
print(z3)
z4 = create_list(5, 'none')
print(z4)
```

Program output:

```
[1, 1, 1, 1, 1]
[0, 0, 0, 0, 0]
['', '', '', '']
['none', 'none', 'none', 'none', 'none']
```

19.2.6

Editing the list

- when preparing general methods for data processing, it is necessary to decide whether the goal is to change values
- the decision depends on the situation, the scope of the list and the goals of further processing
- the following pair of functions execute the same operation with different effects on the original list

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

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

original = [1, 3, 8, 5, 6]
```

```
print(add1(original, 'x'))
print('original:', original)
print('---')
print(add2(original, 'y'))
print('original:', original)
```

Program output:

```
[1, 3, 8, 5, 6, 'x']
original: [1, 3, 8, 5, 6]
---
[1, 3, 8, 5, 6, 'y']
original: [1, 3, 8, 5, 6, 'y']
```

19.2.7

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 sum_(a, b):
  return a + b

s = sum_(10, 20)
print(s)
x = 10
y = 5
z = sum_(x, y)
print(z)
```

Tuple

Chapter 20

20.1 Tuple

20.1.1

Mutable and immutable

- lists, or collections specifically in Python are generally divided into **mutable** (changeable) and **immutable** (unchangeable)
- a typical mutable structure is **list**
- its non-editable version is **tuple** - n tuple
- it supports virtually all operations as in the list, just without operations that change the contents of the list

20.1.2

Why do I need immutable elements?

After all, they represent lists impoverished by the possibility of change???

- faster list iteration, more efficient work with memory, faster copying
- they are usually used for storing data of different types (name, surname, date of birth, salary), while lists are for similar/same data
- they provide the programmer with the assurance that the data will not be rewritten - some parts of the values should not be changed: e.g. the name and surname representing a person should not allow changing only the first name or only the surname, if a change is made, then it is the change of 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

20.1.3

Tuple

- represents a tuple produced from any iterable sequence
- as with the list, it is an ordered collection of elements
- similar to the case of the **list**
- unlike lists, tuples are immutable = once a tuple is created, it cannot be changed in any way

```
point = (100, -75)
print(point)
print(type(point))
```

Program output:

```
(100, -75)
```

- point is a typical example of tuple
- 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, delete it and create a new one

20.1.4

Creating tuple

- dubious operation, since tuple type variable cannot be changed, but functional

```
a = ()
x = tuple()
print(type(a), a)
print(type(x), x)
```

Program output:

```
 ()
 ()
```

- it makes sense to fill the variable with values
- we use round brackets

```
pointA = (10, 10)
pointB = (10, 10, 10)
```

- a special case is the one-element set
- we write it 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:

```

```

20.1.5

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)
```

20.1.6

Element searching

- **len()** - number of elements
- **in** - determines whether the element is in the list
- **count()** - number of occurrences
- **index()** - position of the 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 nothave')
```

Program output:

```
I have
```

- **count()** also works for list

```
print('value 3', t.count(3),'x')
```

Program output:
```
value 3 3 x
```

```
wanted = 6
try:
  i = t.index(wanted)
  print(wanted,'on the position',i)
except:
  print('not found')
```

Program output:
```
not found
```

20.1.7

Element access and slice

- tuple element is accessed via index

```
t = (1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)
print(t[1])
```

Program output:
```
2
```

- however, an attempt to change it leads to an error

```
t[1] = 7
```

Program output:
```
TypeError
'tuple' object does not support item assignment
```

- slice works in a standard way

```
print(t[1:4])
```

Program output:
```
(2, 3, 4)
```

- however, again, it does not allow editing

```
t[1:3] = (1,2)
```

Program output:
```
TypeError
'tuple' object does not support item assignment
```

20.1.8

Tuple iteration

- by default we iterate through **for**

```
t = (1, 2, 3, 4, 3, 5, 9)
for element in t:
  print(element)
```

20.1.9

Function over the list

- **sum**
- **min**
- **max**

```
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
```

20.1.10

Convert tuple -> list

- 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]
```

- beware, the following notation creates a single tuple element as part of the mutable list

```
l2 = [t]
print(l2)
```

Program output:

```
[(1, 2, 3, 4, 3, 5, 9, 8, 11, 3, 5)]
```

20.1.11

Conversion to tuple

- via the **tuple()** function practically universally

```
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)
```

20.2 Tuple in functions

20.2.1

Multiple assignment

- the assignment of multiple values to multiple variables in one line can be written in only one line

```
a, b, c = 1, 2, 3
print(a, b, c)
```

20.2.2

Return multiple outputs from a function

- Python allows multiple values to be returned from a function
- e.g. the area and perimeter of the rectangle

```
def rect_info(a, b):
  s = a * b
  o = 2 * a + 2 * b
  return s, o

area, perimeter = rect_info(5, 6)
print(area, perimeter)
```

- if we load the values into one variable, we have an immutable tuple object at our disposal
- we can execute all the operations allowed for tuples with it

```
result = rect_info(2, 4)
print(result)
print(type(result))
```

Program output:

```
(8, 12)
```

20.2.3

Find all the divisors of the entered 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 print them 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())
list_ = divisors(x)
print(list_)
# print(", ".join(list_))
if len(list_) == 2:
  print("prime number")
else:
  print("composite number")
```

Program output:

```
(1, 7)
prime number
```

20.3 Variadic function

20.3.1

Variadic function

- a function with a mutable(changeable) number of arguments
- a certain form is represented by a function that has some parameters set to default values
- unless we specify them when calling the function, the default ones are used

```
def sum_up(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(sum_up(10))
print(sum_up(20, 30))
```

Program output:

```
10
50
```

- we could use 3 or 4 inputs, but with 5 parameters it is no longer possible to match the value to the corresponding variable

```
print(sum_up(1, 2, 3, 4, 5))
```

Program output:

```
TypeError
sum_up() takes from 1 to 4 positional arguments but 5 were
given
```

20.3.2

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 is preceded by the character *
- allows you to get any number of parameters from a function call and iterate in the function

```
def sum_up(*numbers):
  sum_ = 0
  for i in numbers:
    sum_ = sum_ + i
  return sum_

s = sum_up(1,2,3)
print(s)
```

Program output:

```
6
```

- what type is the **numbers** variable in the function?

```
def sum_up(*numbers):
  print(type(numbers))
  sum_ = 0
  for i in numbers:
    sum_ = sum_ + i
  return sum_

s = sum_up(1,2,3)
```

```
print(s)
```

Program output:

```
6
```

- it is **tuple**

20.3.3

Write a function that, for an input list, returns an entered number of random values from it.

- in this case the first parameter will define the number of elements for the resulting tuple and will be followed by an unknown number of elements to choose from

```
import random

def choose(z, *list_):
  elements = ()
  for i in range(z):
    elements = elements +
(list_[random.randrange(len(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:

```
(5, 1, 5)
('d', 'f', 'f')
('b', 'a', 'b')
```

- if we choose as a list of values the list generated by **range()**, a small problem arises:

```
print(choose(3, range(8)))
print(type(range(8)))
```

Program output:

```
(range(0, 8), range(0, 8), range(0, 8))
```

- it is a special element passed to the function as a **range** element
- in order to send the generated values instead, we have to "unpack" it
- again using "*"

```
print(choose(3, *range(8)))
```

Program output:

```
(3, 1, 2)
```

- we can also expand the **list** before sending it to the variadic function

```
print(choose(3, *['Adam','Beta','Cecil','Dana']))
```

Program output:

```
('Dana', 'Cecil', 'Cecil')
```

20.3.4

Print?

- it is a typical variadic function that allows the printout of any number of elements that are practically of 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
```

List of Lists

Chapter 21

21.1 Table

21.1.1

Write a program that loads a list of students' names and their heights. The number of students is entered by the user at the input. Then find the tallest student and print his name and height.

- with current knowledge we can design and create two lists (list or tuple) and fill them with data

```
number = int(input())
students = []
heights = []
for i in range(number):
  name = input("name:")
  students.append(name)
  height = int(input("height:"))
  heights.append(height)

print(students)
print(heights)
```

- if we look at the task from a non-informatics point of view, it is actually a list of students, where each item consists of a name and height
- we create the list for the purpose of further use, and it is not likely that the data in it should change
- we can create a list whose items will be tuples

```
number = int(input())
students = []
for i in range(number):
  name = input("name:")
  height = int(input("height:"))
  pupils.append((name,height))

print(students)
```

- then we will search the list
- we can remember the greatest height, but finding it is not enough to find out the name, because we need to know who it belongs to

21.1.2

Access to items

- if we work with a list, we get to a specific element based on its index

```
students = [('Ivan' , 150), ('Dana', 174), ('Jozef', 178)]
print(students[1])
```

Program output:

```
('Dana', 174)
```

- the result is an element at the entered position, which in this case is an element of type **tuple**
- we could get data from it, e.g. as follow:

```
students = [('Ivan' , 150), ('Dana', 174), ('Jozef', 178)]
slacker = students[1]
print('name:',slacker[0])
print('height:',slacker[1])
```

Program output:

```
name: Dana
height: 174
```

- or we can choose an entry combining the retrieval of the i-th element and the j-th item from its composite content
- entry:

```
height = students[1][0]
```

- gets the items at position 1 from the students list and from the given item (in this case **tuple**) the data located at index 0

```
students = [('Ivan' , 150), ('Dana', 174), ('Jozef', 178)]
print('name:',students[1][0])
print('height:',students[1][1])
```

Program output:

```
name: Dana
height: 174
```

21.1.3

Table

- such a list can be easily imagined as a table

name	height
Ivan	150
Dana	174
Jozef	178
Jana	162
Michal	179

```
students = [('Ivan' , 150), ('Dana', 174), ('Jozef', 178),
('Jana', 162), ( 'Michal', 179) ]
print(students[3][1])
```

Program output:

```
162
```

- the first index specifies the line and the second index the column

21.1.4

So let's finish the task:

- **Write a program that loads a list of students' names and their heights. The number of students is entered by the user at the input. Then find the tallest student and print his name and height.**

```
number = int(input())
students = []
for i in range(number):
  name = input("name:")
  height = int(input("height:"))
  students.append((name,height))
```

- when searching, we will remember the index (line) of the highest height so far
- after the comparison is done, we take the index and print the values for it at position 0 and 1 in the tuple
- we search by going through the list, using **len** or **enumerate**

```
maxindex = 0
for index, student in enumerate(students):  # going through
the list of students
```

```
  # if the value of the student ('jozo',162) is greater than
the value (height) at the maxindex position in the table,
  # which has been found so far as the largest
  if student[1] > students[maxindex][1]:
    maxindex = index
print(students[maxindex])
```

21.1.5

Modified task

Print the names of all the students whose height is the greatest.

- the functions **sum**, **max** and **min** do work for lists, but only at the level of simple lists
- so in the first step we find the maximum height and in the second we print all the students who reach it

```
students = [('Ivan' , 150), ('Dana', 174), ('Jozef', 178),
('Jana', 162), ('Michal', 178)]
maxheight = 0
for student in students:
  if student[1] > maxheight:
    maxheight = student[1]

for student in students:
  if student[1] == maxheight:
    print(student)
```

Program output:

```
('Jozef', 178)
('Michal', 178)
```

- if we wanted to get a list of such students as a **list**, we can also use comprehension

```
maxi = []
[maxi.append(student) for student in students if student[1] ==
maxheight]
print(maxi)
```

Program output:

```
[('Jozef', 178), ('Michal', 178)]
```

21.1.6

List of lists

- **Register the list of employees with their salaries and number of children.**
- **Add a function that will increase the salary by 10 percent.**
- **Add a function that finds the amount of family allowances (€60 per child) and adds it to the list for each employee as a new value.**

name	salary	children	allowances	sum
Ivan	1500	1	60	1560
Dana	1704	5	300	2004
Jozef	1780	1	60	1840
Jana	1620	3	180	1800
Michal	1709	0	0	1709

- we create a list of employees
- in this case, however, we need to be able to change the content of individual items - employees
- therefore we must create individual items as **list**-s

```
employees = [['Ivan',1500,1], ['Dana',1704,5],
['Jozef',1780,1], ['Jana',1620,3], ['Michal',1709,0]]

def increaseby10percent(z):
  for employee in employees:
    employee[1] = round(employee[1] * 1.1,2)

increaseby10percent(employees)
print(employees)
```

Program output:

```
[['Ivan', 1650.0, 1], ['Dana', 1874.4, 5], ['Jozef', 1958.0,
1], ['Jana', 1782.0, 3], ['Michal', 1879.9, 0]]
```

- we have already mentioned that rewriting the value representing a specific element from the list in the cycle does not change the content of the list, only the representing variable
- if it was a simple variable - this fact would apply
- since the **employee** in the cycle is a variable of type **list**, it affects the assignment of the elements of the **employee** list

- adding a new item representing 60 times the number of children can be done by traversing the list or using comprehension

```
def allowances1(z):
  for employee in employees:
    employee.append(employee[2]*60)

allowances1(employees)
print(employees)
```

Program output:

```
[['Ivan', 1650.0, 1, 60], ['Dana', 1874.4, 5, 300], ['Jozef',
1958.0, 1, 60], ['Jana', 1782.0, 3, 180], ['Michal', 1879.9,
0, 0]]
```

```
employees = [employee + [employee[2]*60] for employee in
employees]
print(employees)
```

Program output:

```
[['Ivan', 1650.0, 1, 60, 60], ['Dana', 1874.4, 5, 300, 300],
['Jozef', 1958.0, 1, 60, 60], ['Jana', 1782.0, 3, 180, 180],
['Michal', 1879.9, 0, 0, 0]]
```

- in both cases, when the function is called multiple times, a new item in the list is always created
- the calculation needs to be adjusted so that the item is added only if the number of list elements in the employee is less than 4
- otherwise, we rewrite the value in column 3 - if the number of children changes

```
def allowances2(z):
  for employee in employees:
    if len(employee) < 4:
      employee.append(employee[2]*60)
    else:
      employee[3] = employee[2]*60

allowances2(employees)
print(employees)
employees[1][2] = 8
allowances2(employees)
print(employees)
```

Program output:

```
[['Ivan', 1650.0, 1, 60, 60], ['Dana', 1874.4, 5, 300, 300],
['Jozef', 1958.0, 1, 60, 60], ['Jana', 1782.0, 3, 180, 180],
['Michal', 1879.9, 0, 0, 0]]
```

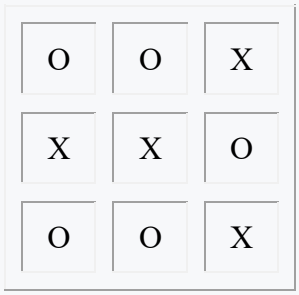
```
[['Ivan', 1650.0, 1, 60, 60], ['Dana', 1874.4, 8, 480, 300],
['Jozef', 1958.0, 1, 60, 60], ['Jana', 1782.0, 3, 180, 180],
['Michal', 1879.9, 0, 0, 0]]
```

21.1.7

Preparing an empty list

- in many cases we need to create a table (list of lists) so that it already has some dimension, but the values in it have been set to the initial value, e.g. empty

Create a tic-tac-toe playing field.



- it is a table with dimensions 3x3, in which there will probably be a list consisting of three empty triples at the beginning

```
line = ['', '', '']
table = [line] * 3
print(table)
```

Program output:

```
[['', '', ''], ['', '', ''], ['', '', '']]
```

- so far everything looks fine
- let's try to enter the first move

```
table[1][1] = 'O'
print(table)
```

Program output:

```
[['', 'O', ''], ['', 'O', ''], ['', 'O', '']]
```

- we are shocked be the result, the element at position 1 in each line has been overwritten
- the reason is the way memory is handled when using a list
- again the same thing applies as in all other cases - the same memory reference is repeated for each line

Solution - eliminate the link between the table and the content of the **line** variable, which can be done in the following ways:

- by entering values

```
table = []
for i in range(3):
  table.append(['','',''])

table[1][1] = 'O'
print(table)
```

Program output:

```
[['', '', ''], ['', 'O', ''], ['', '', '']]
```

- by using a method that returns a new line independent of the previous ones
- a new local variable is always created in its body, which allocates memory in free space

```
def getLine():
  line = ['', '', '']
  return line

table = []
for i in range(3):
  table = table + [getLine()]

table[1][1] = 'O'
print(table)
```

Program output:

```
[['', '', ''], ['', 'O', ''], ['', '', '']]
```

21.1.8

Is function

- the **is** function helps with memory sharing problems
- it identifies whether two objects are the same, whether one list **is** the same as the other

```
line = ['', '', '']
table = [line] * 3
table[1][2] = 'X'
print(table)
print(table[0] is table[1])
```

Program output:

```
[['', '', 'X'], ['', '', 'X'], ['', '', 'X']]
True
```

21.2 Matrix

21.2.1

Matrix

- a matrix is a structure that we have imagined as a table
- it can be multidimensional, most often we encounter two-dimensional one
- it consists of m lines and n columns, with the number of columns in each line being the same

$$\begin{bmatrix} 1 & 3 & 7 \\ 8 & 5 & 0 \\ 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} 2 & 16 & -6 \\ 8 & -4 & 10 \end{bmatrix}$$

- the tic-tac-toe board was a 3 x 3 matrix
- we can create the matrix both at the **tuple** level and at the **list** level

```
tuple_matrix = ((1,0,0),(1,1,0),(0,0,0),(0,1,1))
print(tuple_matrix)
list_matrix = [[1,0,0],[1,1,0],[0,0,0],[0,1,1]]
print(list_matrix)
```

Program output:

```
((1, 0, 0), (1, 1, 0), (0, 0, 0), (0, 1, 1))
[[1, 0, 0], [1, 1, 0], [0, 0, 0], [0, 1, 1]]
```

- a more beautiful (matrix) output will provide us with a line-by-line output with line expansion

```
def output(matrix):
  for i in matrix:
    print(*i)

output(tuple_matrix)
print('--------')
output(list_matrix)
```

Program output:

```
1 0 0
1 1 0
0 0 0
0 1 1
--------
1 0 0
1 1 0
0 0 0
0 1 1
```

21.2.2

Sudoku

Write a program that, given the input 3x3 matrix, evaluates whether it is a correctly solved Sudoku and prints it.

- i.e. each number 1-9 must appear exactly once in the matrix
- we need to determine the method of loading the inputs - ideally, loading by lines, while the numbers will be separated by a space

```
sudoku = []
for i in range(3):
  sudoku = sudoku + [input().split()]
print(sudoku)
```

Program output:

```
 1 2 5  3 5 4 2 8 9[['1', '2', '5'], ['3', '5', '4'], ['2',
'8', '9']]
```

- now we go through the list and find out which number appears in it how many times
- we can create a list in which the number of ones will be in the first place, the number of twos in the second place, etc.
- we only need a list of numbers

```
occurrence = [0] * 10 # we also count with index 0, where the
value will be 0
print(occurrence)
```

Program output:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- we go through the matrix and for each value found we increase the number of its occurrences

```
for line in sudoku:
  for item in line: # the item is a string, we need to change
it to a number
    digit = int(item)
    occurrence[digit] = occurrence[digit] + 1
```

- control, we go through the list and if we find the number of occurrences of a digit being greater than 1, we print it
- if such a situation arises, I will remember it to generate the final answer

```
problem = False
for i in range(1,10):
  if occurrence[i] != 1:
    print(i, '-', occurrence[i], 'times')
    problem = True

if problem:
  print('Wrong entry')
else:
  print('Correct entry')
```

Program output:

```
1 - 2 times
2 - 4 times
3 - 2 times
4 - 2 times
5 - 4 times
6 - 0 times
7 - 0 times
8 - 2 times
9 - 2 times
Wrong entry
```

21.2.3

Addition of matrices

- a typical and simple task
- at the input there are matrixes with the same dimensions
- it is necessary to create a third one, in which the values from the first two, which are located in the same positions, will be added up

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 5 & 0 \\ 7 & 0 & 5 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 2+5 & 3+0 \\ 1+7 & 0+0 & 0+5 \\ 1+2 & 2+1 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 7 & 3 \\ 8 & 0 & 5 \\ 3 & 3 & 3 \end{bmatrix}$$

- for this time we enter the matrix firmly

```
m1 = ((1,2,3),(1,0,0),(1,2,2))
m2 = ((0,5,0),(7,0,5),(2,1,1))
m3 = ()
for i in range(len(m1)):
  line = ()
  for j in range(len(m1[i])):
    line = line + (m1[i][j] + m2[i][j],)
  m3 = m3 + (line,)

print(m3)
```

Program output:

```
((1, 7, 3), (8, 0, 5), (3, 3, 3))
```

- in the first loop we go through the lines, their number represents the number of elements of the tuple in the **m1** list
- in the second loop we go through the number of elements of the inner tuple - these are numbers
- within the loop with the variable **j** we create a tuple for each pair of numbers in the line
- after the end of the **j** loop, we add the created tuple as a new list = line to the resulting matrix
- whether we choose **m1** or **m2** when defining cycles does not matter - the matrices must have the same dimensions - this is how the addition of matrices is defined

21.3 Different number of elements in a line

21.3.1

Different number of elements in a line

- the matrix required to have the same number of elements in each line
- however, in general, there is no such requirement when working with lists
- if we go through the rows, we identify their number in the same way as in the matrix **len(list)**
- if we are traversing the elements in a line, we find out their number via **len(list[i])**, where *i* is the line being traversed

Load individual lines from a text file and create a list containing numbers representing the length of words for each line.

```
Meta received a fine of 265 million euros in Ireland
The concern, which includes the social networks Facebook and
Instagram, for example, violated the data protection law
Read more https://www.sme.sk/
```

Result:

```
4 8 1 4 2 3 7 5 2 7
3 8 5 8 3 6 8 8 3 9 3 8 8 3 4 10 3
4 4 19
```

```
# file preparation
try:
  with open('data.txt','w') as f:
    print('Meta received a fine of 265 million euros in
Ireland', file=f)
    print('The concern, which includes the social networks
Facebook and Instagram, for example, violated the data
protection law', file=f)
    print('Read more https://www.sme.sk/', file=f)
except:
  print('problem with the entry')

# loading a file into a list
list_ = ()
try:
  with open('data.txt','r') as f:
    for line in f:
      list_ = list_ + (line.split(),)
except:
  print('problem')

# traversing the list
lenghts = ()
for i in range(len(list_)):  # traversing the lines
  line_ = ()
  for j in range(len(list_[i])):  # traversing the words
    line_ = line_ + (len(list_[i][j]),)
  lenghts = lenghts + (line_,)

print(list_)
print(lenghts)
```

Program output:

```
(['Meta', 'received', 'a', 'fine', 'of', '265', 'million',
'euros', 'in', 'Ireland'], ['The', 'concern,', 'which',
'includes', 'the', 'social', 'networks', 'Facebook', 'and',
'Instagram,', 'for', 'example,', 'violated', 'the', 'data',
'protection', 'law'], ['Read', 'more', 'https://www.sme.sk/'])
((4, 8, 1, 4, 2, 3, 7, 5, 2, 7), (3, 8, 5, 8, 3, 6, 8, 8, 3,
10, 3, 8, 8, 3, 4, 10, 3), (4, 4, 19))
```

Set and Dictionary

Chapter 22

22.1 Set

22.1.1

A set

- one of the simplest structures designed to hold lists
- working with it is much faster than with other structures, but:
- does not allow storing duplicates
- 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, whose contents can be changed

```
set_ = set()
set_.add(100)
set_.add('Peter')
set_.add(1000.56)
print(set_)
```

Program output:

```
{1000.56, 'Peter', 100}
```

- adding an element with the same content has no effect

```
set_.add('Peter')
print(set_)
```

Program output:

```
{1000.56, 'Peter', 100}
```

- we can check if an element is in the set:

```
a = 'Peter'
b = 'Michal'
if a in set_:
  print(a, 'is in', set_)
else:
  print(a, 'is not in', set_)
print(b in set_)
```

Program output:

```
Peter is in {1000.56, 'Peter', 100}
False
```

22.1.2

Operations on sets

- the easiest is to find the number of elements in a set

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'X', 'Y', 'Z'}
print(len(m1))
print(len(m2))
```

Program output:

```
4
3
```

- in addition to adding, we also need to know how to delete an element
- given the absence of an index, we determine which element to delete

```
m1 = {'A', 'B', 'C', 'D'}
m1.discard('B')
print(m1)
```

Program output:

```
{'C', 'D', 'A'}
```

- if the entered element is not in the set, nothing happens
- if we need information that we are trying to remove a non-existent element, we can use the **remove()** method

```
m1 = {'A', 'B', 'C', 'D'}
m1.remove('B')
print(m1)
m1.remove('B')
print(m1)
```

Program output:

```
{'C', 'D', 'A'}
KeyError
'B'
```

- the complete deletion of elements will be provided by the **clear()** method

```
m1 = {'A', 'B', 'C', 'D'}
m1.clear()
print(m1)
```

Program output:

```
set()
```

22.1.3

Operations with sets

- **update** - inserting elements of another set into a set

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'X', 'Y', 'Z'}
m1.update(m2)
print(m1)
```

Program output:

```
{'D', 'B', 'C', 'X', 'Y', 'A', 'Z'}
```

- **intersection** of sets – identification of elements in both groups

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m1 & m2
print(m3)
```

Program output:

```
{'C', 'A'}
```

- unification of sets

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
m3 = m1 | m2
print(m3)
```

Program output:

```
{'B', 'D', 'C', 'Y', 'A'}
```

22.1.4

Comparing sets

- sets can be compared based on the following rules
- sets are equal if they have the same elements

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'Y', 'C'}
```

```
m3 = {'A', 'B', 'C', 'D'}
print(m1 == m2)
print(m1 == m3)
```

Program output:

```
False
True
```

- one set is smaller than another if it contains all its elements + something

```
m1 = {'A', 'B', 'C', 'D'}
m2 = {'A', 'B', 'C'}
m3 = {'A', 'B', 'Y'}
print(m1 > m2)
print(m1 < m2)
```

Program output:

```
True
False
```

- attention, if the set also contains other elements, then none of them is bigger

```
m1 = {'A', 'B', 'C', 'D'}
m3 = {'A', 'B', 'Y'}
print(m1 > m3)
print(m1 < m3)
```

Program output:

```
False
False
```

22.1.5

Find all the different digits that occur in the entered number.

- we are not interested in how many times they are repeated, only which ones occur there
- so we create a set to which we will add elements
- adding a duplicate element does nothing
- after traversing the number, we will print them

```
number = input()
list_ = set()
for i in number:
  list_.add(i)
print(list_)
```

Program output:

```
 13221354{'5', '1', '3', '4', '2'}
```

22.1.6

Iteration through the elements of a set

- we iterate the same way as in the case of the other lists
- let's print only elements from the set that have more than 4 characters

```
set_ = {'Seven', 'words', 'was', 'found', 'on', 'papyrus',
'scrolls', 'by', 'Black', 'sea'}
for i in set_:
  if len(i) > 4:
    print(i)
```

Program output:

```
Black
words
found
scrolls
Seven
papyrus
```

22.2 Dictionary

22.2.1

Dictionary

- represents a list into which we insert the key and its corresponding value
- subsequently, we can retrieve the value by querying the key
- the order is defined since version 3.7
- it is created using **dict()** or empty parentheses

```
e = {}
d = dict()
print(type(e))
print(type(d))
```

Program output:

```

```

- or possibly by listing values (name, salary)

```
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 is:

- creating a list of unique objects and assign a value to them
- and subsequently, , getting the feature for a unique value corresponding to it
- gaining access to a record known e.g. in php as associative fields
- for unique Adam we get his salary

```
print(d['Adam'])
```

Program output:

```
1500
```

- saveing the necessary parameters describing the object with their values

```
car = {
  "brand": "Ford",
  "type": "Mustang",
  "year": 1964
}

print(car)
print(car["brand"])
```

Program output:

```
{'brand': 'Ford', 'type': 'Mustang', 'year': 1964}
Ford
```

22.2.2

Basic operations

- dictionary size by default trough **len**

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
print(len(d))
```

Program output:

```
4
```

- adding an element - we enter the key value as the index of the element and assign its value

```
d['Zuzana'] = 2100
print(d)
```

Program output:

```
{'Adam': 1500, 'Beta': 1800, 'Cynthia': 1578, 'Damian': 1384,
'Zuzana': 2100}
```

- if we try to get a non-existent element through an index call, an exception is generated

```
print(d['Elvira'])
```

Program output:

```
KeyError
'Elvira'
```

- but if we use the **get()** method, it returns None value in case it does not exist

```
print(d.get('Elvira'))
```

Program output:

```
None
```

- element editing - by assigning a new value to the element defined by the key

```
d['Adam'] = 1999
print(d)
```

Program output:

```
{'Adam': 1999, 'Beta': 1800, 'Cynthia': 1578, 'Damian': 1384,
'Zuzana': 2100}
```

- deleting an element using **del**
- if no such element exists in the list, an exception is generated

```
del d['Adam']
print(d)
```

Program output:

```
{'Beta': 1800, 'Cynthia': 1578, 'Damian': 1384, 'Zuzana':
2100}
```

- an alternative is to use the **pop()** method, which returns the value of the entered key and deletes the given element from the list

- if the element does not exist, an exception is generated

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
name = 'Zuzana'
data = d.pop(name)
print(name, ':', data)
print(d)
```

Program output:

```
KeyError
'Zuzana'
```

- deleting the entire dictionary using **clear()**

```
d.clear()
print(d)
```

Program output:

```
{}
```

22.2.3

Typical examples of use

- only if the key values can be considered unique, e.g.:

```
identification number => name of the person
name of the person => telephone number
municipality => district
telephone number => name of the person
town => GPS coordinates of the center
domain address => IP address
VRN => customer
```

22.2.4

Design a structure that will provide the tracking of borrowed cars in the car rental office.

The car will be identifiable by VRN and the person by name.

- we create a list in which the VRN will be answered by the person who has the car on loan
- one car cannot be borrowed by more than one customer at the same time
- however, one customer may have several vehicles rented at the same time

```
d = {}
d['NR233AA'] = 'Stone'
d['NR258CC'] = 'Plum'
d['NR408XX'] = 'Stone'
d['NR233AA'] = 'Poppy'
d['BA23300'] = 'Plum'

print(d)
```

Program output:

```
{'NR233AA': 'Poppy', 'NR258CC': 'Plum', 'NR408XX': 'Stone',
'BA23300': 'Plum'}
```

- we can print the name of the person who has a rented car with a given VRN at any time

```
car = input()
print(d[car])
```

Program output:

```
 NR258CCPlum
```

- checking whether the given car is in the list
- respectively whether a pair is defined for the key value

```
if "NR233AA" in d:
  print("we have")
```

Program output:

```
we have
```

22.2.5

Traversing a list

- we iterate through keys, values, or dictionary elements

```
d = {'Adam':1500, 'Beta':1800, 'Cynthia':1578, 'Damian': 1384}
print(d.keys())
print(d.values())
print(d.items())
print('--------')
for i in d.keys():
  print(i)
print('--------')
for i in d.values():
```

```
  print(i)
print('--------')
for i in d.items():
  print(i)
```

Program output:

```
dict_keys(['Adam', 'Beta', 'Cynthia', 'Damian'])
dict_values([1500, 1800, 1578, 1384])
dict_items([('Adam', 1500), ('Beta', 1800), ('Cynthia', 1578),
('Damian', 1384)])
--------
Adam
Beta
Cynthia
Damian
--------
1500
1800
1578
1384
--------
('Adam', 1500)
('Beta', 1800)
('Cynthia', 1578)
('Damian', 1384)
```

22.2.6

Practical use 1

- communication between different systems
- 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'}]
```

- the result of this code is a list of dictionaries, where each dictionary describes a person
- let's print a list of names of registered persons

```
for human in z:
  print(human['name'])
```

Program output:

```
Jozef
Klement
Jana
Juan
```

- let's finish the task, find out how many men and how many women we have

```
men = 0
women = 0
for human in z:
  if human['gender'] == 'man':
    men = men + 1
  else:
    women = women + 1
```

```
print('men:',men)
print('women:',women)
```

Program output:

```
men: 3
women: 1
```

22.2.7

Practical use 2

- a dictionary is also mentioned as a typical example of the dictionary
- at the input we have pairs of words in different languages, one of them is chosen as key value, the other as value
- then we will load a word from the user and use it as a key value to quickly find an equivalent

```
dictionary = { 'car' : 'auto', 'student' : 'študent', 'school'
: 'škola', 'street' : 'ulica'}
wanted = input()
try:
  print('translation "' + wanted + '" - ',dictionary[wanted])
except:
  print('not found')
```

Program output:

```
 cartranslation "car" -  auto
```

- the translation in the opposite direction is not so simple
- we have to iterate over the key-value pair and write the key for the found value

```
dictionary = { 'car' : 'auto', 'student' : 'študent', 'school'
: 'škola', 'street' : 'ulica'}
wanted = input()
for key, value in dictionary.items():
  if value == wanted:
    print(value, 'the translation is', key)
print('end')
```

Program output:

```
 auto
auto the translation is car
end
```

PRISCILLA

priscilla.fitped.eu