FITPED | AI

Python - Introduction

Ján Skalka
Ľubomír Benko
Vaida Masiulionytė-Dagienė

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 | AI

Python - Introduction

Published on

November 2024

Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Vaida Masiulionytė-Dagienė | Vilnius University, Lithuania

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 978-80-558-2222-8

TABLE OF CONTENTS

Python Language

Chapter 1

1.1 Python language

📖 1.1.1

1989 is considered to be the year of the birth of the Python language. It was introduced by Guido van Rossum, who for many years played a decisive role in deciding Pythons direction (he gave up leadership in 2018).

Python is a high-level language, meaning it uses selected natural (English) language words to express commands. Its keywords, commands and control structures were designed to match the user's mindset on the one hand and the requirements of an algorithmic language on the other.

The translation of commands to a lower level that the computer understands is realized only when the program is started.

📝 1.1.2

Writing a program in Python consists of writing commands in the English language.

- Yes
- No

📖 1.1.3

Python is an **interpreted programming language**. We cannot usually run the program we write in it by double-clicking on the icon, but we need a translator - interpreter to run it. The interpreter evaluates and executes commands step by step, line by line, command by command.

The interpreter is part of the development environment in every basic installation. With interpreted source code, we have to remember that running a program doesn't mean it's error-free, just that they haven't been discovered yet.

Errors appear only when the interpreter comes to a line of code with an error notation. Therefore, it may happen that the program you create will be functional at the beginning, but an error and an error message will appear only during the program's execution.

📝 1.1.4

Can a program written in **Python** be executed even if there are errors in it?

- Yes
- No

📖 1.1.5

The Python language allows you to write cross-platform applications, which means that the same program code can be run on portable devices, home computers (regardless of the operating system), supercomputers, or even various hardware toys or microcomputers.

Thanks to this, we only need to learn the language once and we can create applications for practically every type of device.

📝 1.1.6

What types of devices can Python be used on?

- microcomputers
- supercomputers
- smartphones
- personal computers
- laptops

📖 1.1.7

When writing the source code, it is necessary to remember that Python is a so-called case sensitive, that is, it is necessary to distinguish between upper and lower case letters. A language interpreter would evaluate the words

```
Command, command, COMMAND
```

or other versions of the entry as different entries.

📝 1.1.8

Select for command

```
Print
```

options that will certainly perform the same operation.

- Print
- print
- PRINT
- PrinT

📝 1.1.9

Choose the correct statements about Python.

- A program written in Python cannot run on different platforms.
- A program written in Python does not pre-find all the errors in the code before it runs.
- Python is case sensitive in source code.
- Python does not have a built-in development environment, so we need to install an external application additionally.

1.2 Development environment on the computer

🕮 1.2.1

As part of the course, we will not need any additional installations or settings, the web browser will take care of everything. We will be using **Python** version 3.

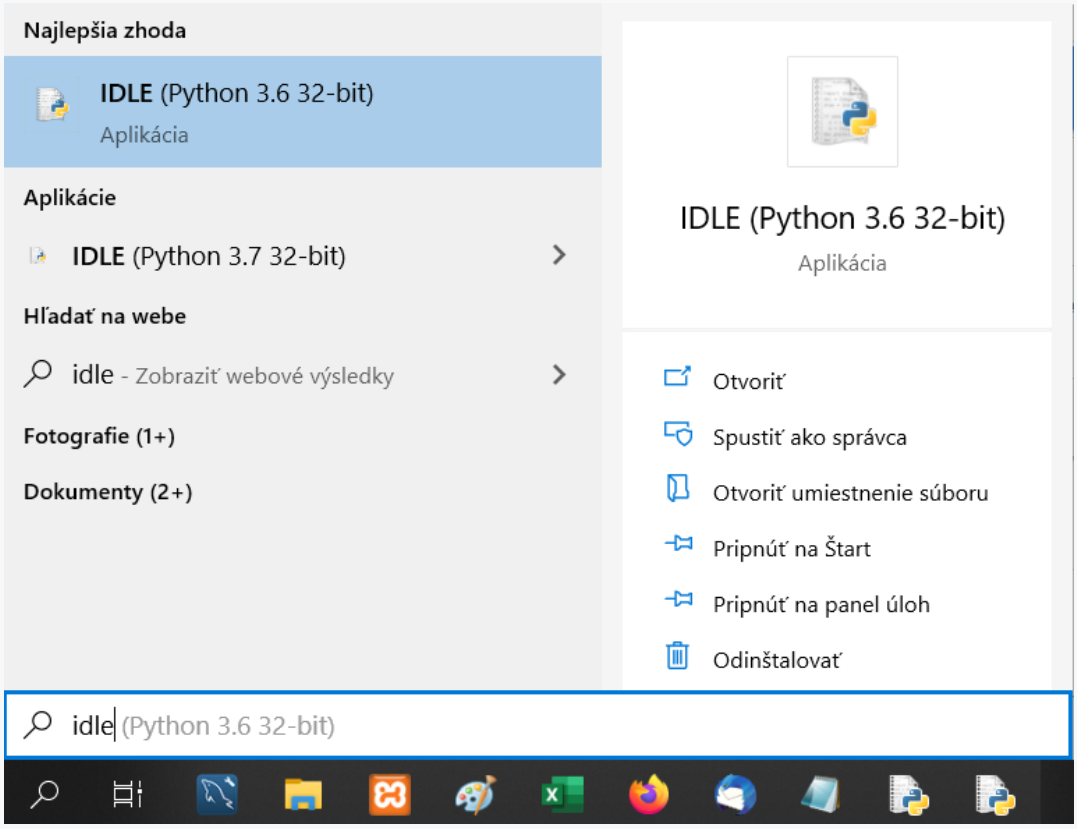
However, considering that our goal is to learn to write programs in **Python**, it would be convenient to install a simple development environment on your computer that will allow you to test the operation and outputs of the programs.

The current version of **Python** is available at http://python.org.

To work in **Python** on a local computer, it is necessary to select and start one of the programming environments (shells), for example the **IDLE** environment, which contains a set of tools to facilitate writing and running programs.

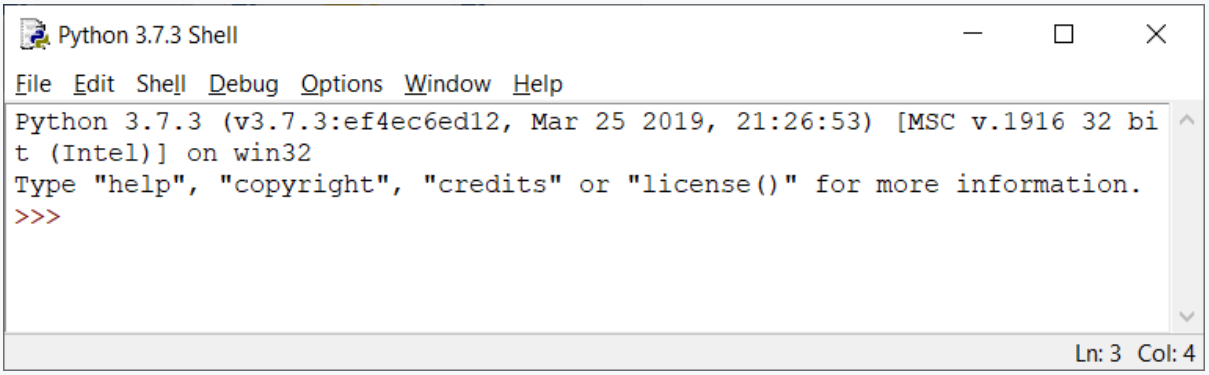
🕮 1.2.2

After installation, we find the **Idle** program in the system and start it. Usually, the link to it includes the information that it is **Idle Python**.



1.2.3

After starting, a simple window will appear with information about the running environment, which includes a menu that allows you to create and run programs.



An open window provides access to the **Python** interpreter where code can be **entered and executed directly**, but this method is not used when writing longer programs.

Example of typing commands into the command line.

```
>>>
```

Type the following command and press **Enter**.

```
print("Welcome to the world of Python")
```

Pressing **Enter** created a **Python** program that displays the greeting "*Welcome to the world of Python*".

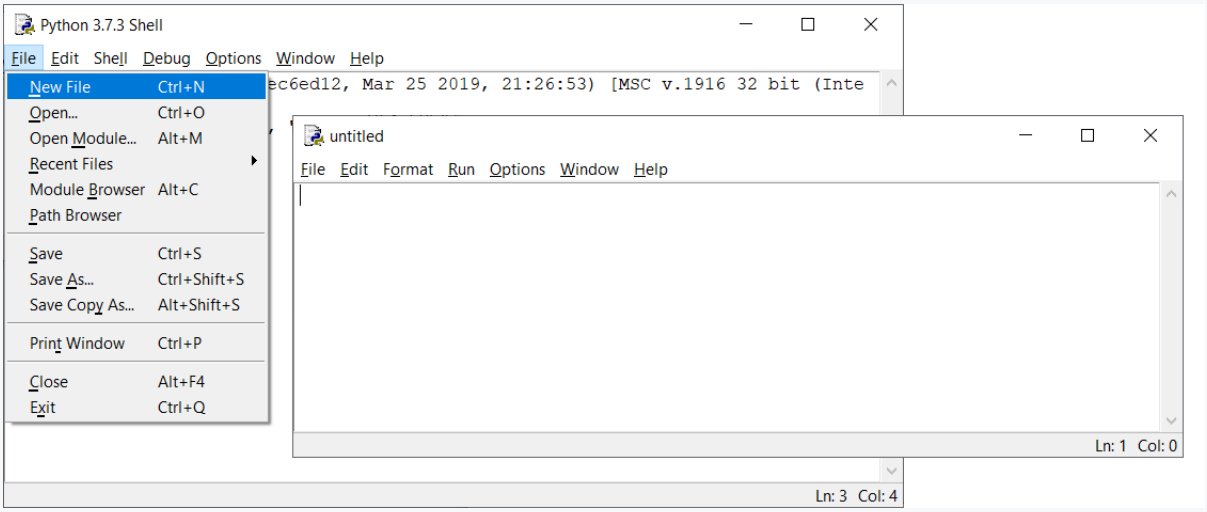
The **print** command ensures that the text is printed on the screen.

📖 1.2.4

As a rule, we save the programs we create in separate files.

To create a new file, we use the command in the menu **File -> New File** by default.

This command will create an empty file into which we can write a program that can be run later.



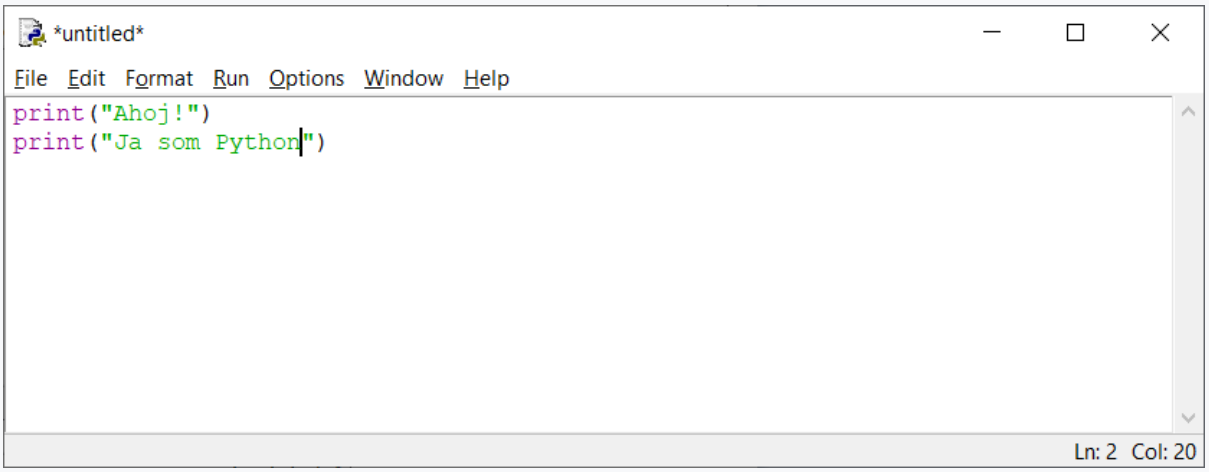
📖 1.2.5

Let's start with a very simple program in which we print some text using the print() command.

We put the text we want to print between quotation marks or apostrophes and write it in parentheses.

Our first program might look like this:

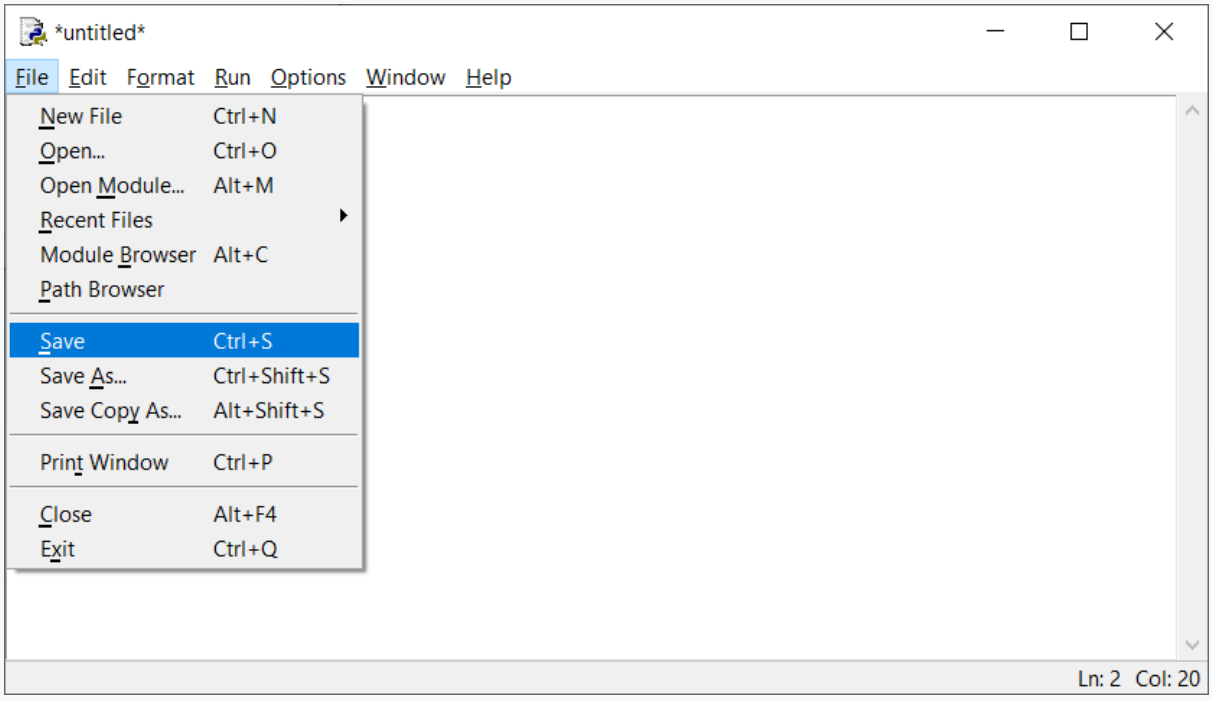
```
print("Hi!")
print("I am Python")
```



1.2.6

The program must be saved before running. We can do this manually, or the environment will force us to do it automatically at the first start.

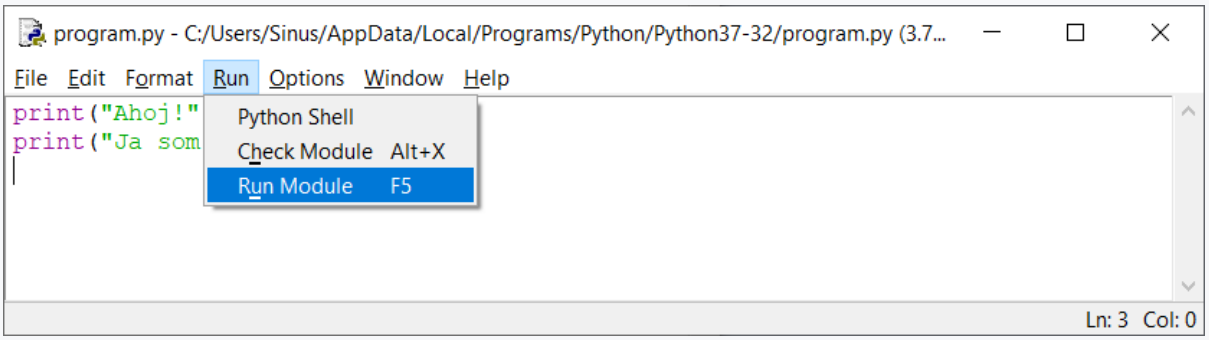
The option **File -> Save** is used for saving.



1.2.7

We start the program via the command **Run -> Run module** or by pressing the **F5** key.

After the first save (or entering the name of the file with the program), the program is saved automatically before starting.



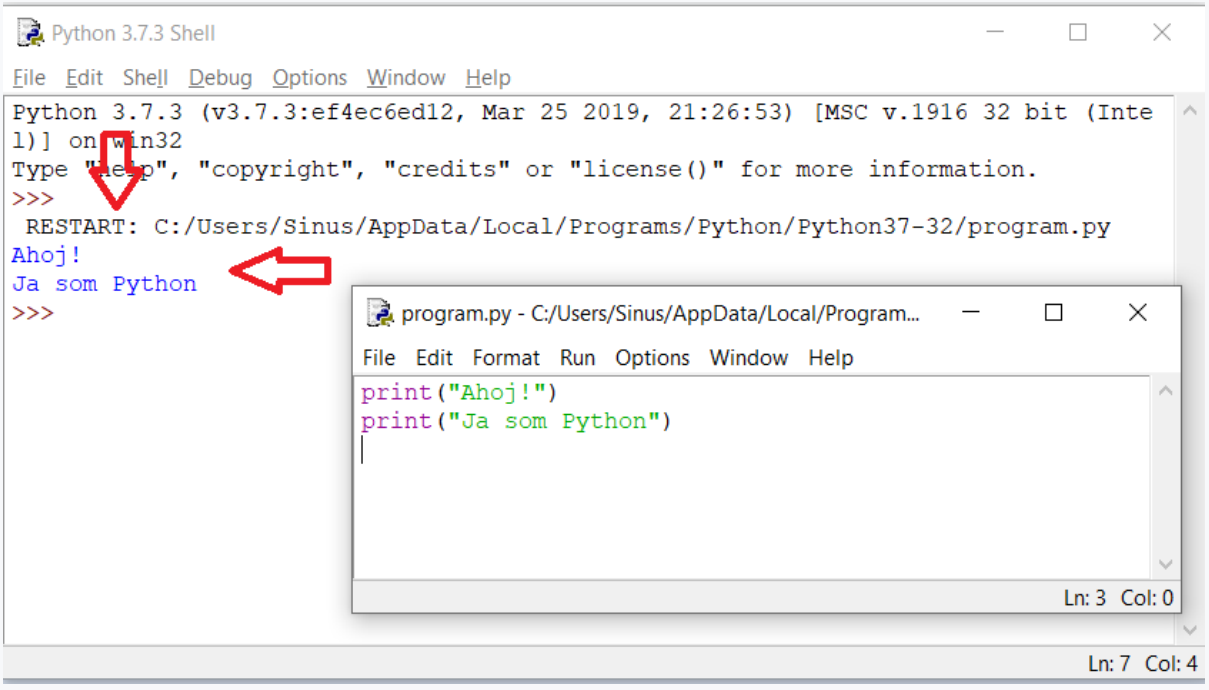
1.2.8

We can see the result of running the program in the main window.

First, information about the start of the program is displayed (with the name under which we saved it), and then the program starts

```
RESTART + program name
```

and then it executes the program's commands - in our case, it prints the two texts that we assigned to it.



The **>>** sign, which appears under the program output after all commands have been executed, indicates that the program has finished its activity.

Output Command

Chapter 2

2.1 Print command - print()

📖 2.1.1

The print() command is used to print text or for executing a simple calculation.

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

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

The command prints on the screen

```
Hello world!
```

In the case of calculation, we write "example input" in brackets:

```
print(16 + 5)
```

and the programming language processes the request by printing the result:

```
21
```

📝 2.1.2

What does the following command print?

```
print("Hello.")
```

- Hello.
- "Hello."
- Hello
- ("Hello.")

📝 2.1.3

Complete the program that prints the text **Python is great.**

```
_____("_____ _____ _____.")
```

- Python
- English
- print
- is
- great

📖 2.1.4

When using the **print()** command, a new line is set for further output after the text in quotes is printed. Therefore, the next **print()** command always prints its text on a new line.

For example, a sequence of commands

```
print("Hello world!")
print("I am a programmer.")
```

prints two independent texts below each other.

```
Hello world!
I am a programmer.
```

📝 2.1.5

Arrange the commands in the program so that the output has the following form.

```
Hello.
I am Priscilla.
I believe,
I will teach you
TO PROGRAM
```

- print("I will teach you")
- print("I am Priscilla.")
- print("I believe,")
- print("TO PROGRAM")
- print("Hello.")

📖 2.1.6

To display the output with multiple values in one line, you need to separate the values with a comma.

```
print("Hello world!", "I am a programmer.")
```

The sequence of commands written in this way prints two independent texts in one line one after the other. A space will be automatically inserted between the texts.

```
Heloo world! I am a programmer.
```

📝 2.1.7

Complete the print() command so that the printed texts are put in one line:

```
Hello. Today is a beautiful day. The sun is shining.
```

Program:

```
print("Hello."_____ "Today is a beautiful day."_____ "The sun is shining.")
```

📝 2.1.8

Complete the program so that it prints the text **Happiness will come on Tuesday**.

```
_____________________________________________________
```

- Wednesday
- .
- "will come"
- "Tuesday"
- "Happiness"
- ;
- "on"
- .
- ,
- print
- ,
-)
- (
- ;
- ;
- .
- ,

🕮 2.1.9

Text in quotation marks, or any sequence of characters or numbers that we will treat as written text (not a number) is referred to as a **string** in the programming language.

The easiest way to identify a string is by placing it between enclosing characters, quotation marks ("") or apostrophes ('').

📝 2.1.10

Which statements are true?

- the string represents a sequence of characters enclosed by e.g. quotation marks
- "mum has emma" is a string

- 603 is a string
- 2 is a string

📖 2.1.11

Python supports two ways of entering strings - enclosing them with quotation marks " or apostrophes '.

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

What is it good for?

If we decided to print the text that contains quotation marks, for example:

```
Ja som "programátor".
```

then we would have a problem, because the entry

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

would be invalid. **Python** would only evaluate the text "I am" as the end of the string and analyze the rest as a new string and look for, for example, a comma as a separator.

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

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

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

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

📝 2.1.12

Which of the codes are written incorrectly?

- print("I am "Peter"!")
- print("I am 'Peter"!')
- print("I am 'Peter'!")
- print('I am "Peter"!')

2.1.13

Complete the correct combinations of quotation marks or apostrophes:

```
print("I am _____very clever'. I hit myself in the forehead
with a hammer._____)
print('In direct speech, sentences are enclosed by a sign
__________);
```

2.2 Simple calculations

2.2.1

The **print()** command provides us with versatile functionality. If we use it without content, it will print, or skip an empty line.

For example:

```
print('Hello')
print()
print('Let's count...')
```

print:

```
Hello
--- nothing is here - just an empty line ---
Let's count...
```

so:

```
Hello

Let's count...
```

2.2.2

Arrange the lines so that the output looks like this:

```
Hello,

The print command can be used for a variety of things:

printing a text
mathematical calculation
inserting empty lines
```

- print()
- print('The print command can be used for a variety of things::')
- print('inserting empty lines')
- print('Hello,')
- print()
- print('printing a text')
- print('mathematical calculation')

📖 2.2.3

The print command can be used not only for text output, but it can also execute various calculations, e.g.

```
print(16+7);
```

first finds the result of the calculation in parentheses and then prints it.

We enter the calculation without quotation marks, on the basis of which the system knows that it is supposed to work with the contents of the brackets as with numbers and we don't just want to print it in the same form as it is in quotation marks.

For the entry

```
print("16+7");
```

the result would be identical to the text in quotation marks:

```
16+7
```

📝 2.2.4

What does the following command print?

```
print(22+17)
```

📝 2.2.5

What does the following command print?

```
print("18+9")
```

📝 2.2.6

What does the following command print?

```
print(22 + 17 + 3)
```

📖 2.2.7

The basic mathematical operations we use in expressions are

- + for addition, e.g. 10 + 20 (we already know that)
- - for subtraction, e.g. 20 - 8,
- * for multiplication, e.g. 5 * 8, the multiplication sign is represented by an asterisk.

The same rules apply to the use of parentheses in expressions as in mathematics, i.e. calculation in parentheses takes precedence over multiplication and division, and these take precedence over addition and subtraction.

Therefore:

```
print(1 + (4 - 1) * 3 + 2 * 8)
```

is calculated:

- first we find the result of the calculation in parentheses and multiply it by the value 3 - calculations in parentheses take precedence,
- then we multiply **2** and **8**,
- finally, we add the obtained values.

So the result is 26.

📝 2.2.8

What does the following command print?

```
print(22 - 17 + 3)
```

📝 2.2.9

What does the following command print?

```
print(3 * 2 - 2)
```

2.3 Programmer's comments

📖 2.3.1

In programs, we often need to note something down, explain, write a note, organize thoughts or add comments for later understanding, or for another user or

programmer. Such text is not intended for the program interpreter and must be ignored, **the code will not be executed, as it will only serve as a comment to the rest of the code.**

We define the information that the text is supposed to be considered a **comment** with the sign #.

The text that is listed after this character in the given line is ignored - if the # character is listed at the beginning of the line, the entire line is logically ignored. The informal rule is that there is a space after the # sign.

E.g.

```
# a line for the program output follows
print('Hello world.')
```

or

```
print('Hello world.')          # this line printed a greeting
print('You are so nice today.')   # this line tried to flatter
me
```

2.3.2

Add characters for comments to the code:

```
_____ this program will print important informations
print('Attention.')
print('Hello world.') _____ first important information
print('You are so nice today.')  _____ second important
information
```

2.3.3

Using the comment character, we can only make a one-line comment. If we want to comment several lines, we must repeat the character in each line.

```
# this is the first line of the comment
# this is the second line of the comment
```

The second type of comments in **Python** are **block comments**, which can contain several lines of code. Such a comment begins and ends with the character """ or '''.

```
"""This is a block comment
        it can contain several lines."""

'''This is also a block comment
```

```
        it can also contain several lines.'''
```

2.3.4

What character is used to insert a multiline comment?

- '''
- #
- //

2.4 My first programs

2.4.1 My first statement

Create a program that prints the following text

```
I use the print command.
```

2.4.2 Triangle

Write a program that creates the following triangle from the "o" characters:

```
o
oo
ooo
oooo
```

2.4.3 Adam and Eva

Create a program that prints the text:

```
I am 'Adam'. Nice to meet you "Eva".
```

Variables

Chapter 3

3.1 Variable

📖 3.1.1

In the first chapter, we directly wrote out information in the form of texts. We didn't remember any information, so we couldn't work with it in several parts of the program.

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

Variables represent a separate place in computer memory where some value can be remembered for later use. We can use one or a large number of variables in the program. In order to distinguish between them, each variable must have its own unique name set by the programmer.

The name of the variable can be practically arbitrary, it is necessary to observe only a few rules, which we will mention later.

Variable values can change during program execution.

📝 3.1.2

Is the following statement true?

A variable in a program can have any name that is determined by the programmer based on the rules defined in the given programming language.

- Yes
- No

📖 3.1.3

A variable **is created** by executing an **assignment command** the moment we first insert a value into it. An assignment command consists of a variable name on the left side, the "**=**" operator, and a value (or a calculation that yields a value) on the right side.

Do premenných budeme spočiatku vkladať číslo alebo text.

```
n = 10
name = 'python'
```

How will it look in the computer? Executing the command creates a new variable based on the name entered on the left side and inserts (assigns) the value entered on the right side into it.

So the name of the variable is a kind of reference to a specific value. In Python, it is not necessary to indicate in advance whether we will store a number, characters, or a sequence of characters - strings. Python takes care of setting the variable correctly automatically. A variable cannot exist without a value.

The Python language interpreter does not care which of the following notations we use

```
n = 10
n=10
n= 10
```

spaces are simply ignored.

> However, there are rules for formal arrangement of the source code (PEP8, https://www.python.org/dev/peps/pep-0008/), which prefer notation where there is one space on either side of the assignment statement.

3.1.4

Complete the program so that the value **25** is inserted into the **temperature** variable.

```
temperature _____ 25
```

3.1.5

Is the following statement true?

In the assignment command, the value located on the right is stored in the variable with the name on the left. The parts of the assignment statement are joined by the "=" sign.

- True
- False

3.1.6

The name of the variable can be anything, if we follow a few rules for naming variables:

The first character of the name must be:

- **Alphabet letter** (lowercase or uppercase). Letters from the alphabet of different languages can also be used (but this is not recommended).
- **The underscore character "_".**

The rest of the variable name can consist of letters, underscore "_" and numbers.

Allowed variable names are e.g.

```
number
_number
Number
Number_1
number_Second
```

Unallowed variable names are e.g.

```
4pieces
#number
```

According to the PEP8 rules, the recommendation for variable names is to use lowercase letters and the underscore character.

3.1.7

Can the word **1class** be a variable name?

- Yes
- No

3.1.8

In variable names, the compiler distinguishes between lower and uppercase letters, therefore the variables **myvariable** and **Myvariable** are two different variables.

3.1.9

Are the letters used in variable names case sensitive in **Python**? If not, the variable "**NUMBER**" and "**number**" represent the same variable.

- The variable name is case sensitive.
- The variable name is not case sensitive.

3.1.10

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

The list of 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		

In the case of incorrect use of the variable name, the translator reacts e.g. with the following error:

```
>>> True = 3
SyntaxError: you cannot assign a keyword
```

3.1.11

Which of the names can be used as a variable name?

- _4num
- num
- num4
- n_u_m
- #num
- 5num
- print
- !warning

3.2 Operations with variables

3.2.1

We already know how to store a value in a variable. We can, of course, also look at the value stored in the variable.

We will use the well-known **print()** command, which, in addition to printing the text in apostrophes or quotation marks, can also print the contents of the variable. We just need to put its name in parentheses.

```
print(variable)
```

We do not enclose the variable name in quotes or apostrophes. Based on the use of these characters, the translator knows whether to print the content that is in the variable or the text that we enclosed in quotes.

```
temperature = 33
print(temperature)   # prints 33 - the value that is stored in
the temperature variable
print("temperature") # prints the text temperature
```

3.2.2

Complete the program so that it prints the value of the variable **v**.

```
v = 5
_____(_____)
```

- print:
- print
- v=
- v
- "v"

3.2.3

We usually don't use variables to just read and print a value, but we can also store the result of a calculation in them. The notation of a calculation is referred to as an **expression**.

For example in the program

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

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

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

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

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

We can print the result stored in the **x** variable

```
print(x)
```

The difference between directly printing the result of the calculation using

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

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

⌨ 3.2.4 Mathematical calculation

Write a program that stores the result of the following mathematical operation in a variable

```
5 + 48 + 3 * 11 - 85
```

and subsequently prints its content using the **print** command.

📝 3.2.5

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

```
c = 15 - 8 * 3
print(c)
```

📝 3.2.6

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

```
a = (10 - 7) * 2 + 4 * 3 - (7 - 2)
print(a)
```

🕮 3.2.7

In addition to the expression consisting only of numerical values (e.g. 5 * 7 + 3), we can also use variables in the expression on the right side.

For example in the program:

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

we first insert values into the **x** and **y** variables.

Subsequently, the calculation will take place based on the expression x + y, where instead of the variables, their values are used - that is, we perform the sum of the **values** that are stored in the variables x and y, i.e. 10 + 20.

Finally, the result is inserted into the variable whose name is given on the left side, i.e. variable **z**.

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

```
z = x + y
```

Before the assignment itself, the right side of the assignment command is always evaluated first, where:

- **x** is replaced by the current value of the variable (read from the corresponding memory location),
- the current value in the variable **y** is added to it,
- the result is stored in the new variable **z**.

And finally, we can print the result

```
print(z)
```

3.2.8

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

```
a = 10
b = 25
c = a + b
print(c)
```

3.2.9 Sum of variables

Write a program that:

- creates a variable **a** and assigns the value 10 to the variable **a**
- creates a variable **b** and assigns the value 17 to the variable **b**
- print the sum of these two variables

3.2.10 Product of two variables

Write a program that:

- creates a variable **a** and assigns the value 5 to the variable **a**
- creates a variable **b** and assigns the value 4 to the variable **b**
- creates a variable **c** and assigns it the product of the variables **a** and **b**
- prints the contents of the variable **c**

file1.py

```
#!/usr/local/bin/python
# create a variable a and assign the value 5 to it
```

```
# create a variable b and assign the value 4 to it
# create a variable c and assign the product a, b to it
# print the contents of the variable c
```

3.3 Variables in expressions

📖 3.3.1

We know that 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 put in its place. However, we can freely combine directly entered values and variables in expressions, e.g.:

```
amount = 100
new_amount = amount - 20
```

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

📝 3.3.2

What value will be stored in the variable x after the execution of the following commands?

```
a = 10
z = 15
x = a + 15 + z
```

- 40
- 30
- 15
- 0

📝 3.3.3

What value will be stored in the variable **c** after the calculation is executed?

```
a = 2
b = 3
c = 2 * (a + b) - a * 3 + b
```

- 7
- 18
- 1
- 10

📖 3.3.4

We can often encounter an entry 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-hand side of the assignment commandis evaluated first:

- the current value of the **poc** variable is read,
- the value 1 is added to it,
- the result of the expression is then stored in the **poc** variable, rewriting its original value.

So the value 3 is printed first, it changes and the value 4 is printed in the second printout.

📝 3.3.5

Arrange the assignment command so that the program prints 5 and 8 below each other.

```
a = 1
b = 2
```

- print(b)
- a = a + b
- b = 2 * a + b
- a = b - a
- print(a)

📖 3.3.6

Just as we could execute the calculation when printing values, we can also execute it with variables and send not only the values of the variables in the print command, but also the results of the operations to the output.

Therefore instead of:

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

we can omit the calculation of the variable c and directly print the sum of the contents of the two variables.

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

3.3.7

What does the following program print?

```
a = 3
b = 4
c = 5
print(a + b * c)
```

3.3.8 Calculations with variables I.

Write a program that:

- creates a variable **a** and assigns the value 15 to the variable **a**,
- creates a variable **b** and assigns the value 40 to the variable **b**,
- creates a variable **c** and assigns it the difference of the variables **a** and **b**,
- prints the contents of variable **c**,
- creates a variable **d** and assigns to it the difference of the product of **a** and **b** with the contents of the variable **c**
- prints the contents of variable **d**,
- prints the sum of variables **c** and **d**.

file1.py

```
#!/usr/local/bin/python
# create a variable a and assign the value 15 to it
# create a variable b and assign the value 40 to it
# create variable c and assign it the difference of variables
a and b
# print the contents of the variable c
# create a variable d and assign to it the difference of the
product of a and b with the contents of the variable c
# print the contents of the variable d
# print the sum of the variables c and d
```

3.3.9 Calculations with variables II.

Write a program that:

- creates a variable **a** and assigns it 333,

- creates a variable **b** and assigns it a value 203 smaller - use the calculation,
- prints the value of **b**,
- creates a variable c and assigns the product of **a** and **b** to it,
- creates a variable **first** and assigns it the value **c**,
- changes the contents of the variable **first** by multiplying it by two (puts its double into the variable **first**),
- creates a variable **second** and assigns it the value of the variable **first**
- changes the content of the variable **second** by subtracting the product of **a** and **b** from its original content,
- prints the difference of the variables **first** and **second**.

file1.py

```
#!/usr/local/bin/python
# create variable a and assign it 333,
# create a variable b and assign it a value 203 smaller - use
the calculation,
# print the value of b,
# create a variable c and assign to it the product of a and b,
# create the variable first and assign it the value c,
# change the content of variable first by multiplying it by
two (puts double of it into variable first),
# create a variable second and assign it the value of the
variable first
# change the content of the variable second by subtracting the
product of a and b from its original content,
# print the difference of the first and second variables
```

3.4 Output formatting

🕮 3.4.1

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

```
10 20 30
```

while the space is filled automatically with the **print** command.

3.4.2

What does the following program print?

```
a = 15
b = 10
print(a, b, a * b - a)
```

- 15 10 135
- 135
- 15, 10, 135
- 10, 15, 135

3.4.3

Variables **a**, **b** have set initial values. What is printed after the sequence of commands is executed?

```
a = 10
b = 5

a = a + b
b = a - b
a = a - b

print(a, b)
```

- 5 5
- 0 5
- 5 10
- 10 5

3.4.4

Let's imagine a program

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

whose output takes the form

```
10 20 30
```

However, such output is very brief and the sequence of numbers in the printout may be unclear to the user. Therefore, it is advisable to combine the printout of variables with descriptive texts that explain the numbers in more detail, e.g.

```
x = 25
print('The value of the variable x is', x)
```

prints:

```
The value of the variable x is 25
```

Note that the variable **x** mentioned in apostrophes is an ordinary string of characters, so no value is substituted for it. The value is only inserted into the stand-alone variable name specified as the second parameter of the **print** function.

3.4.5

What does the following program print?

```
a = 15
b = 10
print('The result of the sum of a and b is', a + b)
```

- The result of the sum of a and b is 25
- The result of the sum of a and b is25
- The result of the sum of 10 and 15 is 25
- The result of the sum of 10 10 15 is 25

3.4.6

If we want to print several variables in combination with several static texts, we must separate each part of the output 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 output, do not forget that there should be a space between the individual parts of the printout - this is added automatically in the printout thanks to the rules defined for the **print** command.

3.4.7

Complete the code correctly so that we receive the exact required out.

```
j = 25
k = 12
print(_____, '__________, __________ '=', _____ - _____)
```

The required output is

```
25 - 12 = 13
```

- j
- k
- k
- +
- j
- -
- ,
- j
- '
- k

3.4.8

Choose the correct statement.

- A variable in Python can hold any value, but once it is assigned it must not change.
- A variable is created after the program is started and is valid throughout the entire program's run.
- A variable in Python is created after assigning a value to it, and we can use its value repeatedly in the following part of the program.

3.4.9 Printing the values of variables I.

Write a program that:

- creates a variable **a** and assigns the value 10 to the variable **a**,
- creates a variable **b** and assigns the value 20 to the variable **b**,
- prints the product of the variables a and b in the form:

```
The product of 10 and 20 is 200
```

file1.py

```
#!/usr/local/bin/python
# create a variable a and assign the value 10 to it
# create a variable b and assign it a value of 20
```

```
# print the product of a and b in the form "The product of 10
and 20 is 200"
```

⌨ 3.4.10 Printing the values of variables II.

Write a program that:

- creates a variable **a** and assigns the value 37 to the variable **a**
- creates a variable **b** and assigns the value 26 to the variable **b**,
- prints the results of operations with variables in the form:

```
summ: 37 + 26 = 63
difference: 37 - 26 = 11
product: 37 * 26 = 962
```

file1.py

```
#!/usr/local/bin/python
# create a variable a and assign the value 37 to it
# create a variable b and assign the value 26 to it
# provide a printout for the sum
# provide a printout for the difference
# provide a printout for the product
```

Input Command

Chapter 4

4.1 Input command

📖 4.1.1

We usually expect programs to be able to solve the problem for different values.

If we had a program that could only add the values 230 and 180, then instead of writing it, it would be enough to use a calculator or just the knowledge from elementary school.

The purpose of the program is to be able to perform the same operation or sequence of operations with arbitrary values. These must somehow get into the program without us having to write them directly into the code. We refer to them as input values, and in order for the program to work with them, it needs to recieve them from the user and store them in variables.

Operations that provide the loading of values are referred to as **input operations**. Initially, it involves entering the desired values from the keyboard and reading them by the program.

📝 4.1.2

What are the commands that ensure the loading of values from the user to the program called?

- input
- output
- ongoing

📖 4.1.3

So far, we have been working with variables that we have previously set to some specific values. If we wanted to change the inputs, we needed to rewrite the program. However, we cannot expect such an activity from the user of the program, and we must teach the program to read input values from the user.

The **input()** command is used to retrieve data from user input. The command reads the data entered by the keyboard and confirmed by Enter and returns it in the form of a text string. The value returned by the **input()** command can then be stored in a variable.

The entire entry then has the form:

```
data = input()
```

Before stopping the program and waiting for the input, it is usually necessary to inform the user about what we actually expect at the input, e.g.:

```
print('Enter a name: ')
name = input()
```

Subsequently, we can work with such a variable.

4.1.4

Which of the entries for data input is correct if we want to insert the entered data into the variable a?

- input(a)
- a = input()
- input() = a

4.1.5

Instructions for the user can also be entered in the parentheses of the input() command.

```
  name = input('Enter a name: ')
  surname = input('Enter a surname: ')
```

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

```
print('Hello', name, surname)
```

Prints the text according to the entered name and surname, e.g.:

```
Hello Joseph Carrot
```

4.1.6

Complete the program so that it reads the name and surname and outputs:

```
Hello, your name is Ferko Carrot.
name = _____('Enter a name: ')
surname = _____('Enter a surname: ')
print('Hello, your name is', _____, _____)
```

4.1.7 Name and age

Complete the program so that it reads the name and age and outputs:

```
Hello, your name is Ferko and you are 17 years old.
```

When retrieving data, display the texts 'Enter name: ' and 'Enter age: '.

Prepare the solution to the task on the computer in the Idle environment to see how the program works and just copy it here.

For example for entry:

```
Jozef
15
```

the output will be

```
Hello, your name is Jozef and you are 15 years old
```

file1.py

```
#!/usr/local/bin/python
# load name with display text 'Enter name: '
# load age with display text 'Enter age: '
# load the text in the form Hello, you are Jozef and you are
15 so that you use the content in the variables
```

4.2 Not a sum like a sum

📖 4.2.1

Let's try the following commands whose purpose is to add two numbers.

```
a = input('Enter the 1st number: ')
b = input('Enter the 2nd number: ')
print(a + b)
```

For example, if we entered the values 3 and 2, we would expect to see the value 5. However, the result is the value 32.

The reason is that the **input()** command cannot distinguish whether the input is text or number and returns the value in a more universal form - as a text **string**.

Since in **Python** we don't need to define the variables or their type (number or string) in advance and we leave the type determination to the compiler in the first step, getting the correct value can be a bit more time consuming...

📝 4.2.2

What will be the result of the following sequence of commands for input values 5 and 7?

```
a = input()
b = input()
```

```
print(b + a)
```

- 75
- 57
- 12
- 21

⌨ 4.2.3 'Stupid' sum

Write a program that adds two values given as text input (i.e., do not convert them to numbers).

When retrieving data, display the texts 'Enter 1st value: ' and 'Enter 2nd value: '.

Prepare the solution to the task on the computer in the Idle environment to see how the program works and just copy it here.

For example for entry:

```
20
15
```

the output will be

```
2015
```

For entry:

```
mother
winter
```

the output will be

```
motherwinter
```

file1.py

```
#!/usr/local/bin/python
# load the first value with the text display 'Enter 1st value:
', e.g. into the variable first
# load the second value with the text display 'Enter 2nd
value: ', e.g. into the variable second
# print the result as the 'sum' of the first and second values
```

🕮 4.2.4

If we are sure that a number will be entered at the input and we want to work with the read value as a number, we need to **convert** the read data - change it from a

string to an integer. We do so using the **int()** command. So the program will look like this:

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

The **int(text)** entry ensures the transformation of the text into a number and assigning it to the variables a, b inserts the numerical value returned by the int() command.

With such values, the sum operation then executes mathematical addition.

If there is text stored in the variables, the "+" operation will combine them, if there is a number stored in both variables, the "+" operation will perform their mathematical sum.

Attention, if one value is text and the other numeric, the program throws an error:

```
print(text1 + d) --- TypeError: must be str, not int
```

4.2.5

What is the result of the following sequence of commands for input values 5 and 7?

```
first = input()
second= input()
a = int(first)
b = int(second)
print(b + a)
```

- 75
- 57
- 12
- 35

4.2.6 Mathematical sum

Write a program that adds two values entered as numbers on the input (i.e. converts them to numbers after reading them).

When retrieving data, display the texts 'Enter 1st value: ' and 'Enter 2nd value: '.

Prepare the solution to the task on the computer in the *Idle* environment to see how the program works and just copy it here.

For example for input:

```
20
15
```

the output will be

```
35
```

For input:

```
10
-1
```

the output will be

```
9
```

file1.py

```
#!/usr/local/bin/python
# load the first value with the text display 'Enter 1st value:
', e.g. into the text1 variable
# convert the first value to a number, e.g. into the variable
a
# load the second value with the text display 'Enter 2nd
value: ', e.g. into the text2 variable
# convert the second value to a number, e.g. to variable b
# print the result as the sum of the first and second numeric
values  (obtained after conversion)
```

4.2.7

Add commands to the source code so that double of the read value is printed:

```
text = _____()
a = _____(_____)
print(2 _____ _____)
```

- input
- int
- text

- int
- input
- +
- a
- a
- text
- *

4.3 Input (programs)

⌨ 4.3.1 Greeting

Write a program that reads the user's name and then greets him, e.g.:

```
input: Jozef
output: Hello, Jozef
```

Display text 'Enter name: ' when input is loaded

⌨ 4.3.2 Job

Write a program that reads the user's name, his job and then prints the information in the form, e.g.:

```
input:
Jozef
mason
output: Jozef is mason .
```

Display text 'Enter name: ' and 'Enter occupation: ' when input is loaded

file1.py

```
#!/usr/local/bin/python
# load the name with the text display 'Enter name: ', e.g.
into the variable name
# load the second value with the text display 'Enter job: ',
e.g. to the variable job
# print the result in the print command
```

⌨ 4.3.3 Product of two numbers

Write a program that multiplies two input values entered as numbers (i.e. converts them to numbers after reading them).

When retrieving data, display the texts 'Enter 1st value: ' and 'Enter 2nd value: '.

Prepare the solution to the task on the computer in the Idle **environment to see how the program works and just copy it here.**

For example for the input:

```
20
15
```

the output will be

```
300
```

For the input:

```
10
-1
```

the output will be

```
-10
```

file1.py

```
#!/usr/local/bin/python
# load the first value with the text display 'Enter 1st value:
', e.g. into the variable text1
# convert the first value to a number, e.g. into the variable
a
# load the second value with the text display 'Enter 2nd
value: ', e.g. into the variable text2
# convert the second value to a number, e.g. to variable b
# print the result as the product of the first and second
numerical values (obtained after conversion) - using the sign
*
```

⌨ 4.3.4 Sum of three numbers

Write a program that adds three values entered as numbers on the input (i.e. converts them to numbers after reading them).

When using the **input()** command, for the sake of simplifying the check (and unnecessary delay due to typos), we will skip displaying the text for the user:

Don't use the notation:

```
text1 = input('Enter the 1st value:')
```

but the notation

```
text1 = input()
```

Prepare the solution to the task on the computer in the *Idle* environment to see how the program works and just copy it here.

For example for the input:

```
20
15
10
```

the output will be

```
45
```

file1.py

```
#!/usr/local/bin/python
# load the first value, e.g. into the variable text1
# convert the first value to a number, e.g. into the variable
a
# load the second value, e.g. into the variable text2
# convert the second value to a number, e.g. into the variable
b
# load the third value, e.g. into the variable text3
# convert the third value to a number, e.g. into the variable
c
# print the result as the sum of the first, second and third
numerical values (obtained after conversion)
```

⌨ 4.3.5 Tenfold

Load an integer value and print a decuple of it, e.g.:

```
input : 3
output: 30
```

```
input : 5
output: 50
```

⌨ 4.3.6 Family allowances

The state contributes a fixed amount of money to family for each child. Write a code that, for the specified number of children, calculates how much of family

allowances will the given family receive and prints this amount. The allowance for one child is EUR 30.

```
input : 3
output: 90
```

```
input : 8
output: 240
```

⌨ 4.3.7 Garden area

For the specified length of the side of the square garden, calculate and write how much mesh is needed to fence it and what its area is. Also follow the order of outputs - first the fence, then the area, e.g.:

```
input : 3
output:
fence: 12
area: 9
```

```
input : 5
output:
fence: 20
area: 25
```

file1.py

```
#!/usr/local/bin/python
# load text data from input
# convert the text data to a number representing the length of
the side of the garden
# calculate the length of the fence
# print the length of the fence
# calculate the area
# print the area
```

⌨ 4.3.8 Swimming pool

During the construction of the pool, it is necessary to purchase tiles that will be placed on the bottom and side walls that will be spread around the pool.

Tiles are sold by the surface, side walls by the meter. Calculate for the specified width and length of the pool how many m2 of tiles will be needed to cover the bottom and how many m of side walls will be needed to cover the perimeter.

At the input, the width and length are listed in separate lines - load each value with a separate input() command, which always reads the entire content listed in the line.

At the output, first indicate the area of the tiles on the bottom, and then the area of the walls. Separate the values with a space.

```
input :
3
2
output: 6 10
```

```
input :
5
4
output: 20 18
```

file1.py

```
#!/usr/local/bin/python
# load a text data representing the width
# load a text data representing the length
# convert data to integer variables
# calculate and print first the area, then the perimeter
```

Conditional Statement

Chapter 5

5.1 Conditional statement

📖 5.1.1

A sequence of commands that are executed in the order in which they are written in the program is called **sequence**.

In such a case, the compiler proceeds by individual commands, and when the command is executed, it proceeds to the next one.

All the programs we have encountered so far worked in just that way, e.g.:

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

📝 5.1.2

What is the name of an order of commands that are executed in the order in which they are written?

- sequence
- consequence
- score

📖 5.1.3

However, most programs do not only contain simple sequences, but very often need to decide how to proceed further based on the processed data. Branching gives us the ability to decide and execute other commands based on whether or not a condition is met.

It consists of a condition and commands that are executed in case of fulfillment and non-fulfilment of the condition.

The branching command has the following form:

```
if condition:
      command
```

e.g.:

```
if age < 10:
   print("minor")
if age > 18:
```

```
    print("adult")
```

etc.

The basis of branching is an **if** statement followed by a condition that must result either 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. These commands must be offset from the margin by the same number of spaces.

> The size of the offset is not strictly given. However, according to PEP8, 4 spaces are recommended.

Command, or commands are executed only if the condition is met.

If the condition is not met, the command is not executed and the program continues with the next command.

5.1.4

Complete the program with a conditional statement so that if the value of the age variable is greater than 18, the text: "adult" is printed.

```
txt = input()
age = int(txt)
_____ age > 18_____
    _____("adult")
```

5.1.5

In the previous program, we gave an example of incomplete branching, when we processed only the situation when the condition was met. Quite often, however, we need to treat both situations - when the condition is met and when the condition is not met. Then we talk about full branching.

In this case, we add an **else** part to the original form of the command. The command then has the form:

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

e.g.:

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

Both the **if** and **else** statements must have the same offset - in this case they start from the left margin.

If the value of the **age** variable is greater than 18, the text "adult" is printed, otherwise (that is, the age is less than or equal to 18) the text "minor" is printed.

5.1.6

Complete the program so that if the value of the variable **height** is less than 160, the text: "small" is printed, otherwise "big".

```
txt = input()
height = _____(txt)
_____ height < 160_____
    print("small_____)
__________
    print("big")
```

- ;
- '
- :
- esle
- fi
- else
- "
- :
- int
- if
- ;

5.1.7

The part of the program that is executed when the condition is met is called the **positive branch**, the part of the program that is executed when the condition is not met, the **negative branch**.

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

5.1.8

Arrange the lines of source code whose results is the output of the larger number from the given pair **a** and **b**.

```
a = 20
b = 30
```

- if a > b:
- print(b)
- print(a)
- else:

5.1.9

What are the parts of a conditional statement containing the statements to be executed if the condition is met or not met called?

- branches
- conditions
- brackets

5.1.10

Using one command in the positive and one in the negative branch is rather exceptional, we usually need to use more commands. The fact that several commands are to be executed in a certain branch is provided by an offset - it defines the entire block of commands.

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

Commands 1-3 are executed if the condition is met, commands 4-5 are executed if the condition is not met.

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

5.1.11

Arrange the source code lines so that commands 5 and 3 are executed if the condition is met, and commands 1, 2, and 7 are executed otherwise. Let commands 4 and 6 be executed after the branch is processed.

- command4
- command2
- command1
- command6
- else:
- command3
- if condition:
- command7
- command5

5.2 Taks with a condition

5.2.1

So far, we have only used the greater or lesser sign in the condition. However, we can also compare using other signs:

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

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

5.2.2

Complete the code with the correct characters for comparison:

```
txt = input()
a = int(txt)
if a _____ 0:
    print("A zero value was entered")
else:
    print("A non-zero value was entered")
print("end")
```

📖 5.2.3

Let's test the values of two variables and print whether they are the same or which one is greater. We actually need to test three options.

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

Let's try it through a simple **if** statement.

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

📝 5.2.4

Complete the code that determines whether the value stored in the **a** variable is positive, negative, or zero.

```
_____ a _____ 0:
    print('zero value')
_____ a > 0:
    print('_____')
_____ a < 0_____
    print('_____')
```

- positive number
- negative number
- if
- ->
- if
- else
- else
- =
- ==
- if
- :

⌨ 5.2.5 Positive/Negative

Write a program that will print whether the given integer is positive or negative. For the purposes of this task, consider zero as a positive number.

```
Input : 1
Output: positive
```

```
Input : -3
Output: negative
```

```
Input : 0
Output: positive
```

file1.py

```
#!/usr/local/bin/python
# load a value and convert it to a number
# process the comparison whether the value is positive
# print that it is a positive number
# otherwise print that it is a negative number
```

⌨ 5.2.6 Comparison of numbers

Write a program that, given two given numbers, prints the greater of them. If the numbers are equal, print "numbers are equal".

```
Input :
3
2
Output: 3
```

```
Input :
2
8
Output: 8
```

```
Input :
2
2
Output: numbers are equal
```

file1.py

```
#!/usr/local/bin/python
# load the values and convert them to numbers
# process the comparison and print the result
# process a and b are equal
# process a is greater than b
# process b is greater than a
```

⌨ 5.2.7 Part-time job

Write a program that, for a given age, will display whether the given person can work part-time (can/cannot). A person who is at least 17 years old can have a part-time job.

```
Input : 1
Output: cannot
```

```
Input : 17
Output: can
```

```
Input : 105
Output: can
```

⌨ 5.2.8 The absolute value of a number

```
Write a program that prints the absolute value of a given
integer.
Input : 0
Output: 0
```

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

```
Input : -8
Output: 8
```

⌨ 5.2.9 Comparison of numbers II.

Write a program that, for two given numbers, finds (and stores in the variable **max**) the larger of them. If they are equal, an arbitrary one of them will be printed. Ensure the result by printing the contents of the variable **max**.

```
Input :
3
2
Output: 3
```

```
Input :
2
8
Output: 8
```

```
Input :
2
```

```
2
Output: 2
```

5.3 Multiple conditional statement

🕮 5.3.1

Although the previous solution is correct, it makes sense to consider another one as well.

```
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)
```

However, if we look at the code through the eyes of an experienced programmer, it is inefficient - even if the values are identical and we already have a result after evaluating the first condition, other conditions are tested unnecessarily.

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

```
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)
```

This is the correct solution, the condition a > b is tested only if a == b does not hold. And the condition a < b is tested only if neither a == b nor a < b applies.

However, the entry can be shortened, because we do not have to execute the last test. Validity takes effect automatically.

```
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)
```

5.3.2

Complete a more efficient solution to find out what value is stored in the variable a.

```
_____ a _____ 0:
    print('zero value')
_____:
    _____ a _____ 0:
        print('positive number')
    _____:
        print('negative number')
```

- if
- >
- =
- if
- else
- if
- else
- else
- else
- ==

5.3.3

In addition, Python provides an option to shorten this notation as well. There is a version of the **if - elif - else** command for successive evaluation of several conditions.

In case the compiler evaluates any of the conditions as fulfilled, it no longer evaluates the following **el-if** branches and 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 need to be specified.

In the optimal entry, our solution would look as follows.

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

5.3.4

Complete the code.

```
a = input ('Enter a number')
b = input ('Enter another number')


_____
  print ('Entered numbers are the same.')
_____
  print ('Entered numbers are not the same.')
```

5.3.5

Complete the solution using the **if-elif-else** structure.

```
_____ a _____ 0:
    print('zero value')
_____ a _____ 0:
    print('positive number')
_____:
    print('negative number')
```

- else
- if
- ==
- elif
- >

5.3.6

What is printed after the program is executed?

```
x = 5
if (x == 5):
    print('Hi')
    print('Hello')
```

```
else:
    print('Cheers')
print('Ciao')
```

- 'Hi Hello Ciao'
- 'Hi Hello'
- 'Cheers Ciao'

5.3.7

What is the result of the given code?

```
a = 4
b = 10
if a == 0:
  print(b)
else:
  print(a)
```

- 4
- 10
- 0

5.3.8

Arrange the source code correctly to tell if a number is positive, negative, or zero.

- print(n, 'is zero.')
- if n > 0:
- elif n < 0:
- print(n, 'is negative.')
- n = 10
- print(n, 'is positive.')
- else:

5.3.9 Test results

Write a program that, given the average result from the test and the number of points you have achieved, will print whether you have achieved an above-average, average or below-average result. The first input value is the average, the second is the achieved result.

```
Input:
10
12
Output: above-average
```

```
Input:
20
18
Output: below-average
```

```
Input:
33
33
Output: average
```

⌨ 5.3.10 A well-deserved salary

Write a program that, given the average salary and your salary, will print out whether you have above-average, average or below-average earnings and by how much. The first input value is the average salary, the second is your income.

```
Input:
1000
1200
Output: 200 above-average
```

```
Input:
2000
1600
Output: 400 below-average 400
```

```
Input:
1333
1333
Output: average
```

⌨ 5.3.11 Maximum of three numbers

Write a program that prints the largest of the three entered numbers. If any numbers are the same, it prints the largest value.

```
Input:
2
4
6
Output: 6
```

```
Input:
2
1
```

```
2
Output: 2
```

⌨ 5.3.12 Maximum of four numbers

Write a program that prints the largest of the 4 entered numbers. If any numbers are the same, it prints the largest value.

```
Input:
2
3
4
6
Output: 6
```

```
Input:
2
2
1
2
Output: 2
```

file1.py

```
#!/usr/local/bin/python
#!/usr/local/bin/python
# load four values and convert them to numbers
# declare the first number as max - put a in the max variable
# if the second number is greater than max, store the value of
the second number in max
# if the third number is greater than max, store the value of
the third number in max
# if the fourth number is greater than max, store the value of
the fourth number in max
# print the result
```

Loop

Chapter 6

6.1 Commands repetition

📖 6.1.1

Very often we need to repeat part of the algorithm. A record that enables repetition is referred to as a **loop**. For each repetition, it is important **what** (body of the loop) is to be repeated and **when, for what values,** the commands in the body of the loop are to be executed.

A loop allows a part of a program to be repeated for a given list of values or until a condition is met, e.g.:

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

```
for values:  for coins 10,20,50,10,10
what:                add to purse
```

```
until when:     while there is something on your account
what:           buy presents
```

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

📝 6.1.2

How do we call an entry in a program that allows repetition of actions?

- loop
- branching
- sequence

📖 6.1.3

Print the text 'Python' 7 times below.

To complete this task, we need to **repeat** the printout of the value 7 times:

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

The same effect can be achieved by using a loop, which allows the repeating part of the program (in this case, the printout) to be written into the program only once.

In the repetition definition, we need to specify a group of values for which the printout is supposed 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.

6.1.4

How many times will the following loop be executed?

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

6.1.5

The number of repetitions of the loop does not depend on the values that are listed in the group, but on their number.

For example we can provide the task **Write "Hello" 5 times below** with the following program:

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

Even in this case, the variable gradually acquires 5 different values and for each of them it prints the text "Hello" once. However, the value of the variable i is not used anywhere.

A loop will fulfill the same role

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

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

6.1.6

How many times will the following loop be executed?

```
for i in 2, 4, 0, 10, 20, 10:
   print(i)
```

📖 6.1.7

In general, the definition of a loop takes the form:

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

For everything to work correctly, it must be true:

1. a sequence must be defined as a sequence of values that we can traverse,
2. there must be a colon at the end of the first line of the loop,
3. commands to be executed must be offset from the margin.

What is a sequence of values? For now, we just need to know that they are comma-separated values. Each value has its place in the sequence and for the previous case we needed it in the form of 1, 2, 3, 4, 5, 6, 7.

📝 6.1.8

Which of the following lists can be used to print the greeting 4 times?

```
for i in ??????:
   print("Good morning")
```

- 1, 2, 3, 4
- 8, 9, 8, 11
- 0, 1, 2, 3
- 1, 1, 2, 3, 4
- 0, 0, 0, 0, 4
- 0, 4

⌨ 6.1.9 Hello

Write a program that prints the word "Hello" 10 times below

```
Output:
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

6.2 Enumerated values

📖 6.2.1

So far, we have only used the loop to print the same content. However, it also allows the use of values that are entered sequentially in its header.

Print the values 1-6 below.

To complete this task, we need to **repeat** the value printout 6 times:

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

If we want to simplify the task using the loop, it is enough to specify the sequence of values that the loop is supposed print in its definition:

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

The loop works in such a way that the variable **i** acquires the value of the first element of the sequence and the command is executed, in which 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.

📝 6.2.2

What values will the following cycle print?

```
for i in 8, 7, 3, 1, 5:
    print(i)
```

- 8 7 3 1 5
- 8 7 6 5 1
- 5 4 3 2 1
- 1 2 3 4 5

📖 6.2.3

Write a program that prints 5 lines with the text "I know how to use the loop now." and in each of them it displays 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.
```

The program is relatively simple: we already know that we need to print the values 1-5. So we insert the sequence 1-5 into the definition of the cycle and ensure that the variable also acquires these values gradually.

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

The action to be repeated consists of printing the variable **i** using the command

```
print(i)
```

by which we would achieve a printout of values 1-5 below each other.

It is enough for us to add the desired text, which is **unchanging**, to the **changing** numbers:

```
print(i, 'I know how to use the loop now.')
```

The loop with the output will therefore have the form:

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

⌨ 6.2.4 Output with serial number

Write an algorithm that prints "Hello" 10 times to the console in the form "Hello - 1 x" and "Hello - 2 x" in the next line... "Hello - 10 x".

```
Output:
Hello - 1 x
Hello - 2 x
Hello - 3 x
Hello - 4 x
Hello - 5 x
Hello - 6 x
Hello - 7 x
Hello - 8 x
Hello - 9 x
Hello - 10 x
```

📖 6.2.5

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

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

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

For the output, we need to ensure the following form (e.g. for input 5):

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

In the first line there is a 1-fold, in the second a double, in the third a triple, etc.

The loop with the output will therefore have the form:

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

In this loop, we print the value from the listed sequence and its **a**-multiple.

📝 6.2.6

Complete the program that prints their double for numbers 5-10 in the form:

```
5 - 10
6 - 12
etc.
```

```
for i _____ 5, _____:
    print(i, _____, _____)
```

- 5-10
- '-'
- i * i
- '-'
- in
- 6-10
- i * 2
- 6..10
- i
- 5..10
- i ++ i

- 6, 7, 8, 10
- 6, 7, 8, 9, 10

⌨ 6.2.7 Square

Write a program that for numbers 5-10 prints their square (**a * a**) in the form:

```
5 - 25
6 - 36
etc.
```

📝 6.2.8

Complete the program that prints a small multiplication table for the entered number in the following form:

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

```
text = _____('Enter an integer from 1 to 10: ')
a = _____(text)
for i in 1, 2, 3, 4, _____, _____, 7, 8, 9, _____:
    print(_____, '_____', _____, '_____', _____)
```

- 5
- input
- *
- =
- i * a
- 9
- input
- i + i
- +
- a
- a
- 0
- i * i
- i
- 5
- 6
- get
- .
- 10
- int
- i

📖 6.2.9

Modify the program with the multiplier so that the rows are separated from each other by a line:

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

The loop with the output has the form:

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

to which we need to add the command with the printout "lines". This command is supposed to be repeated after each row of numbers, so it should be part of the loop.

We will add it with the same offset as the command that prints the numbers.

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

By setting the same offset of the commands in the loop, we say that all of them should 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 offset.

For example the entry:

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

would end by printing all multiplications and add a "line" at the end.

The entry

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

would result in an error because it is not possible to determine which part of the program the second print() command belongs to.

⌨ 6.2.10 Carpet with pattern I.

Write a program that 'weaves a patterned carpet' - it will alternately write lines with 10 "o" characters and 10 "x" characters.

```
oooooooooo
xxxxxxxxxx
oooooooooo
xxxxxxxxxx
etc.
```

Let 20 lines be printed in total.

⌨ 6.2.11 Carpet with pattern II.

Adjust the previous program that 'weaves a patterned carpet' so that after every pair of lines with 10 "o" and "x" characters, it adds a line with '-'.

```
oooooooooo
xxxxxxxxxx
----------
oooooooooo
xxxxxxxxxx
----------
oooooooooo
etc.
```

Let 30 lines be printed in total.

6.3 Generated range

🕮 6.3.1

So that we don't always have to manually write the range of values, just imagine how long it would take us to list e.g. 100 values, Python offers a number list generator.

Python uses the **range()** command to generate a list of numbers.

The simplest version of *range()* is to enter a single parameter - the number of integers (how many numbers from zero will be in the list).

For example

```
range(3)
```

returns the list of **numbers**

```
0, 1, 2
```

which we can traverse right away.

Using the command itself does nothing, it only creates a sequence, which we can use directly in the loop.

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

prints:

```
0
1
2
```

If we need to repeat an action **n times**, just generate a list with **n** values using **range(n)**.

```
for i in range(n):
```

The list generated in this way always **starts with the value 0** and ends with the value n-1.

6.3.2

What number will be the first element in the list produced by **range(10)**?

6.3.3

What will be the last number of the list generated by the **range(15)** command.

6.3.4

Sometimes we need to generate a list of values that does not start with zero. It is enough to add a second parameter to the **range()** command.

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

- **start** determines the starting element of the sequence
- **stop** determines a stop for us, the final element of the sequence, the same applies as in the simpler version, that this element is no longer inserted into the list.

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

prints the sequence:

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

6.3.5

Complete the parameters of the **range()** command so that the given list is printed.

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

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

6.4 Stepping in range()

6.4.1

In the **range()** command, it is also possible to define a step by which the values in the list will change by leaps.

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

The step of the change is determined via the third parameter - **step**.

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

prints the sequence

```
10, 15, 20, 25, 30
```

> The same rules still apply to the **start** and **stop** parameters - the last value of the sequence is **at least** 1 less than the **stop** value.

6.4.2

Complete the parameters of the **range()** command so that the given list is printed.

```
5, 9, 13, 17, 21, 25
```

```
for i in range(_____, _____, _____)
```

```
    print(i)
```

- 26
- 4
- 3
- 6
- 5
- 25
- 4
- 31

6.4.3

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

```
5, 9, 13, 17, 21, 25
```

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

- 26
- 27
- 28
- 29
- 25
- 30
- 24

6.4.4

Let's modify our multiplication table problem by using the automatically generated list.

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

for i in range(1, 11):
    print(i, '*', number, '=', i * number)
```

6.4.5

Complete the code so that the loop prints the squares of the numbers from 1 to 15.

```
_____ i _____ range(_____, _____)_____
      print('the square of', i, 'is', _____)
```

- :
- for
- i + i
- 0
- of
- 1
- i * i
- 15
- 16
- in
- -

📖 6.4.6

The **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 **start** and **stop** values, **start > stop** must apply in this case. The value specified in the stop will no longer appear in the list.

```
range(20, 10, -3)
```

creates:

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

📝 6.4.7

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

```
z = range(100, 0, -10)
```

📝 6.4.8

Complete the program so that it prints even numbers from 20 to 40.

```
for i in range(_____,_____,_____):
    print(i)
```

- 39
- -

- 40
- 19
- 41
- 2
- 21
- 0
- 2
- 20
- 1

⌨ 6.4.9 Multiplier

Write a program that prints a small multiplier for a given integer.

```
Input : -5
Output:
1 * -5 = -5
2 * -5 = -10
3 * -5 = -15
4 * -5 = -20
5 * -5 = -25
6 * -5 = -30
7 * -5 = -35
8 * -5 = -40
9 * -5 = -45
10 * -5 = -50
```

```
Input : 5
Output:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

⌨ 6.4.10 Multiples of three

Write a program that, for two given numbers, prints the triples of all the numbers between them. If the first number is greater than the second, print them in reverse order.

```
Input :
5
8
Output:
15
18
21
24
```

```
Input :
1
-2
Output:
3
0
-3
-6
```

Operations in a Loop

Chapter 7

7.1 Sum in the loop

7.1.1

What does the following sequence of commands print?

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

- 0 5 1 5 2 5 3 5 4 5
- 0 1 2 3 4 5
- 0 5 1 5 2 5 3 5 4 5 5 5
- 1 5 2 5 3 5 4 5 5 5
- 1 5 2 5 3 5 4 5 5
- generates an error

7.1.2

What does the following sequence of commands print?

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

- 0 5 1 5 2 5 3 5 4 5
- 0 1 2 3 4 5
- 0 1 2 3 4 5 5
- 1 2 3 4 5 5
- 1 2 3 4 5
- generates an error

7.1.3

What does the following sequence of commands print?

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

- 0 5 1 5 2 5 3 5 4 5
- 0 1 2 3 4 5
- 0 1 2 3 4 5 5
- 1 2 3 4 5 5
- 1 2 3 4 5
- generates an error

📖 7.1.4

For the given list of amounts spent when shopping in a shopping cetre find out how much you've spent

Have you spent in individual stores:

```
12, 16, 33, 8, 21, 17
```

Use a loop to find out and print the subtotals after each receipt is added.

Our task is to count the listed values and print subtotals, i.e.:

- after counting the first receipt, the amount will be 12,
- after adding the second 12 + 16 = 28,
- after adding the third 28 + 33 = 61,
- etc. till the end of the list.

Repetition consists in adding individual amounts to the sum determined so far. So we define repetition for individual values as:

```
for i in 12, 16, 33, 8, 21, 17:
```

To store the running result, we will use the variable **total**, which will gradually increase by the amount that follows in the sequence.

```
for i in 12, 16, 33, 8, 21, 17:
   total = total + i
```

Before we use the variable **total** to add the first value, we need to determine the initial value. Logically, it has the value 0 before counting in the first receipt:

```
total = 0
for i in 12, 16, 33, 8, 21, 17:
   total = total + i
```

In the task, there also is a request to print a running value, so we add:

```
total = 0
for i in 12, 16, 33, 8, 21, 17:
   total = total + i
   print(total)
```

To show how the loop is progressing and how the values of individual variables change, a trace table is used, which lists the values of the variables used at each step of the loop.

```
i     calculation                              total
      before the beginning of the loop          0
12    total = total + 12, i.e. 0 + 12           12
16    sum = sum + 16, i.e. 12 + 16              28
33    sum = sum + 33, i.e. 28 + 33              61
8     sum = sum + 8,  i.e. 61 + 8               69
21    sum = sum + 21, i.e. 69 + 21              90
17    sum = sum + 17, i.e. 90 + 17              107
```

7.1.5

Complete the program to find the number of visitors to the rope park during the Rope Climbing Week.

```
total = _____
for i _____ _____:
    total = _____ _____ i
_____("There was", _____, "visitors.")
```

- total
- *
- input
- 120, 50, 17, 33, 45, 91
- total
- 1
- total
- int()
- 0
- +
- +
- 54, 72, 101, 12, 54, 33, 19
- print
- 2
- total
- in
- i
- i
- 12, 0, 50, 17, 3, 3, 45, 91

7.1.6 Can we buy it all?

Write a program that, given the amount in our wallet and the entered prices of individual pieces of sports clothing, determines whether we can buy all the pieces of clothing. The result of the program will be only a yes/no output.

Load the amount in the wallet with the text: "Enter your amount: ".

The prices of individual pieces of clothing are 120, 50, 17, 33, 45, 91.

E.g. for

```
Input : 100
Output: no
```

```
Input : 1000
Output: yes
```

file1.py

```
#!/usr/local/bin/python
# load the amount in your wallet with the required text
display and convert it to a number
# prepare a variable to store the ongoing amount
# find out the total amount of funds needed
# check whether you have more money than you need
```

🕮 7.1.7

Calculate the sum of the first 100 positive numbers.

Our task is to add the values **1 + 2 + 3 + 4 + 5 + 6 .... + 99 + 100.**

We will add the numbers gradually - in a loop that will repeat from 1 to 100 and we will add each additional value. We will use the variable **sum_** to store the temporary result, which will be incremented by the value stored in i at each step. At the beginning, it is logically empty - it contains the value 0

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

To monitor the activity of the loop, we can create a tracking table.

```
i     calculation                                sum_
       before the beginning on the loop           0
1      sum_ = sum_ + 1, i.e. 0 + 1                1
2      sum_ = sum_ + 2, i.e. 1 + 2                3
3      sum_ = sum_ + 3, i.e. 3 + 3                6
4      sum_ = sum_ + 4, i.e. 6 + 4                10
...
99     sum_ = sum_ + 99, i.e. 4851 + 99     4950
100    sum_ = sum_ + 100, i.e. 4950 + 100   5050
```

7.1.8

Arrange the code so that we get the sum of the first n numbers. First do the conversion to a number then set the variable sum_.

- n = int(text)
- text = input('Enter a number: ')
- print('The sum of first', n, 'numbers is', sum_)
- sum_ = sum_ + i
- for i in range(1, n + 1):
- sum_ = 0

7.1.9 The sum of the interval

Write a program that finds the sum of the numbers between two given values. These are entered at the input by first entering a smaller, then a larger value.

E.g. for

```
Input:
5
7
Output: 18
```

```
Input:
10
80
Output: 3195
```

7.2 Product in the loop

7.2.1

Now let's have a look at tasks requiring multiplication.

Calculate the product of the first 10 numbers.

It is a multiplication of the numbers 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10. In general, we have to do multiplication in sequence just like addition.

> Although we could solve this task by writing the multiplication in a single command, this is more of an exception as we will see later.

If we break down the individual steps of the calculation, then similarly to the previous example, we need some "storage" with a neutral value.

```
product = 1
```

The starting value in the case of multiplication is the value 1. In the case of the sum, it was the value 0, the addition of which did not change the result of the sum, in the case of multiplication it is 1. Indeed, if we used 0 as the starting value, then by multiplying it by any number, we would again only got a value of 0.

So let's first put the value 1 in the variable **product**:

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

If we look at the procedure more carefully, we will find a diagram

```
product = product * i
```

where **i** goes from 1 to 10. Be careful with **range()** we have to start with the value 1, because multiplying by zero would not get us to different value.

And we have a finished program:

```
product = 1
for i in range(1, 11):
    product = product * i

print(product)
```

7.2.2

Complete the code so that it calculates the product of numbers between two values (including threshold values).

```
text1 = _____('Enter smaller value: ')
text2 = _____('Enter larger value: ')
a = int(text1)
b = int(text2)
product = _____
for i in _____(_____, _____):
    product = _____ _____ i
print('The product of numbers from', a, 'to', b, 'is',
product)
```

- a
- product
- b + 1
- a-1
- =*
- p
- 1
- input
- input
- 0
- b-1
- "
- print
- ==
- a+1
- range
- *
- =
- b
- print

⌨ 7.2.3 Factorial

Write a program that calculates the factorial for a given number n (n! = n.(n-1). ... .3.2.1). Print the interim result to the console.

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

```
Input : 4
Output:
1
2
6
24
```

⌨ 7.2.4 Product without multiplication

Write a program that, for two given positive integers, finds the product without using the multiplication operation.

```
Input :
5
3
```

```
Output: 15
```

```
Input :
5
5
Output: 25
```

```
Input :
2
5
Output: 10
```

📖 7.2.5

Multiply the given sequence of numbers. Show the serial number of the numbers in sequence you are multiplying.

For example for the sequence: 4, 9, 2, 5, 2, the output would look like this:

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

In order to be able to display which step we are currently in, we need a counter that we will increase by 1 in each step of the loop. We can name the variable **step**.

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

⌨ 7.2.6 Product of numbers in an interval

Write a program that calculates the product of all integers between two given values. Ensure that the program displays both the sequence number of the multiplication and the intermediate result of the product in the individual steps of the loop during the run.

```
Input :
```

```
5
7
Output:
1 - 5
2 - 30
3 - 210
210
```

```
Input :
2
5
Output:
1 - 2
2 - 6
3 - 24
4 - 120
120
```

7.3 Data loading

📖 7.3.1

Write a program that finds how much you spent on shopping in the last week by asking for the amounts spent for each day.

In this case, we don't know the purchase values in advance to list them in the program as enumerated loop values, but we have to get them from the user after the program starts.

We will use a scheme where the current subtotal will increase with each new value. Let's go step by step. First, we insert a neutral value - 0 into the variable **total**, we haven't inserted any data into the overall package yet.

```
total = 0
text = input('Enter the amount: ')  # we load the amount for
the 1st day
daily_amount = int(text)            # convert it to a number
total = total + daily_amount        # we add the 1nd day
amount

text = input('Enter the amount: ')  # we load the amount for
the 2nd day
daily_amount = int(text)            # convert it to a number
```

```
total = total + daily_amount          # we add the 2nd day
amount

text = input('Enter the amount: ')    # we load the amount for
the 3rd day
daily_amount = int(text)              # convert it to a number
total = total + daily_amount          # we add the 3rd day
amount
```

Two things are important in the following code:

- **several commands (steps) are repeated** - which is not a problem, we can repeat any number of commands in the loop
- we use **the same variable for loading values** from the user (text and daily_amount) - this approach is also standard - we can use the variable multiple times to store the value, of course with the fact that the old value is always overwritten by the new one

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

and we repeat this sequence n times.

After loading and adding all the sums into the storage, we just print the result.

```
total = 0
for i in range(7):                  # a week has 7 days, we
repeat 7 times
    text = input('Enter the amount: ')
    daily_amount = int(text)
    total = total + daily_amount

print(total, 'was spent in a week.')
```

7.3.2

Arrange the lines of the program so that for the given number of students in five 1st year classes, it finds out how many 1st graders attend the school.

- total = total + students
- text = input('Enter the number of students: ')
- print('There is', 'students in the classes.')
- students = int(text)
- for i in range(5):

- total = 0

⌨ 7.3.3 Missed classes

Write a program to find the number of missed classes for the first half of the year. Each of the five months of the school semester is entered separately. Let the output take the form: "X classes were missed."

```
Input:
5
10
12
3
0
Output: 30 hours were missed.
```

```
Input:
2
2
1
0
3
Output:
8
```

🕮 7.3.4

Get the temperatures for the last n days and calculate their average value. Let the temperatures be integer values.

In this case, we do not know in advance not only the temperature values, but also their number. In order to load them, we first need to know their number, which we will then use as the number of loop repetitions.

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

The average is calculated as sum / count. The number of values is loaded at the beginning of the program, the sum will be increased continuously by loading temperatures on individual days.

```
sum_ = 0
for i in range(n):
    text = input('Enter the temperature: ')
    t = int(text)
    sum_ = sum_ + t
```

Finally, to determine the average, we divide the sum of temperatures by their number - we use the "/" operator for division.

```
print('The average temperature in', n, 'days is', sum_/n)
```

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

```
5.0
```

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

7.3.5

Complete the missing code so that it calculates the grade point average on the report card.

```
text = input('Enter the number of grades: ')
number = _____(text)
sum_ = _____
_____ i in _____(_____):
    text = input('Enter a grade: ')
    g = int(text)
    sum_ = sum_ + _____

print('The grade point average is', _____/_____)
```

- sum_
- number
- zz
- for
- number
- 1
- z
- range
- input
- int
- z
- number
- cycle
- sum_
- pp
- 0

⌨ 7.3.6 Average weight of students in class

Write a program that finds the average weight of the students in the class. In the first step, find out the number of students, in the second, load their weights, and finally, in the third, calculate and write the average.

```
Input: 4
50
55
54
60
Output:
54.75
```

```
Input: 5
66
82
58
60
71
Output:
67.4
```

7.4 Loop (programs)

⌨ 7.4.1 The sum of the first n numbers

Write a program to find the sum of the first **n natural** numbers, the number of which is given in the input. Print the interim results as well.

```
Input: 5
Output:
1
3
6
10
15
```

```
Input: 4
Output:
1
3
6
10
```

⌨ 7.4.2 Product without multiplication II.

Write a program that finds the product of two given integers (even negative) without using the multiplication operation.

```
Input :
-5
3
Output:
-15
```

```
Input :
-5
-5
Output:
25
```

```
Input :
2
5
Output:
10
```

Data Types

Chapter 8

8.1 Data types

📖 8.1.1

Every variable we used so far was defined by a pair:

- **the name** by which we refer to it,
- **data type** that determines whether it stores text or a number (these two types of values are enough for us for now).

When we loaded bdata from the input, we always got it in text form, and if we wanted to work with it as with numbers, we had to **convert** it.

📝 8.1.2

Complete the function to convert input from text to integer.

```
aa = input()
a = _____(aa)
```

📖 8.1.3

As we have already mentioned, **Python** is a language with dynamic type checking, i.e. there is no need to define data types for variables in advance. Based on the values, Python itself determines what type of data it is working with when it is needed.

Some of the most commonly used data types are

- **integer** - integers, e.g. 10 or 5,
- **float** - floating point numbers, used to store decimal numbers, e.g. 10.5 or 5.3,
- **string** - string represented by apostrophes or double quotes, e.g. "Python" or 'Python'.

When writing decimal numbers, we use a decimal point - in this case the comma is evaluated as an error.

```
x = 4     # 4 - will be interpreted as 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 data type string.
```

📝 8.1.4

For the following assignments, determine the correct data type of the variables

```
a = 'Anna'   # _____
b = 3.15     # _____
c = 4        # _____
d = 'x'      # _____
e = 5.0      # _____
f = '1987'   # _____
g = 2010     # _____
```

- float
- string
- integer
- string
- integer
- integer
- integer
- string
- float
- string
- string
- float
- float

📖 8.1.5

For every variable or expression, we can find out its data type. The **type()** function is one of Python's built-in functions that can be used to return information about the type of a value.

For example, by executing the following commands:

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

the result is returned

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

which means that the entered value is treated as an integer.

By executing the commands:

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

the result is returned

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

which means that the entered value is treated as a string.

We can also execute the entry

```
print(type("Python"))
```

which, however, does not really make sense to use in this form, because at first glance we can see that it is a string.

8.1.6

Complete the code that prints the data type of the variables:

```
a = 22.1
print(_____(a))
b = 'The mare has a small hip.'
print(_____(b))
```

8.1.7

What does the following command print?

```
print(type(3.5))
```

- <|class 'float'>
- <|class 'int'>
- <|class 'str'>

8.1.8

What does the following command print?

```
print(type(input('Enter a number: ')))
```

- <|class 'str'>
- <|class 'int'>
- <|class 'float'>

8.2 Conversion

8.2.1

By acquiring a value, the data type of the variable is determined. In order to perform operations on the content of variables intended for another data type, we need

operations that can transform data between individual types. Python supports three basic converting functions that convert one data type into another data type:

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

The following conversions are supported:

```
int(10.5)       # returns 10
int('69')       # returns 69
float(100)      # returns 100.0
float('35.53') # returns 35.53
str(24)         # returns '24'
str(66.99)      # returns '66.99'
```

8.2.2

What value will be stored in variable x after the conversion?

```
x = str(3.14)
```

- "3.14"
- 3.14
- "3,14"
- 3,14

8.2.3

If the conversion operation fails, the compiler displays an error message.

E.g. for:

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

is displayed

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

informing about an unacceptable character, or string.

8.2.4

Which of the conversions end successfully?

- a = int(3.14)

- a = float(3.14)
- a = float('3.14')
- a = int('3.14')
- a = int(3,14)

📖 8.2.5

For two values stored in integer variables, let's get the notation of their subtraction and store it in the variable **result**.

For example for 8 and 5 we get the result

```
8 - 5 = 3
```

We will use the addition operation to join the strings, the program will look like this:

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

From the integer form to the **string** type, we use the **str()** function, which can be used both for a variable and for the result of a mathematical operation.

📝 8.2.6

Complete the code so that the result of the program is an output

```
data 10
```

```
p = 'data'
result = p + '_____' + _____(10)
print(result)
```

📝 8.2.7

By completing the given command, change the entered value from string type to integer type.

```
a = '10'
b = _____a_____ + 3
```

📝 8.2.8

What will be the result of the given code?

```
p = input("Enter a number")
```

```
10 + p
```

- TypeError: unsupported operand type(s) for +: 'int' and 'str'
- '10+entered value'
- 10+entered value

Numeric Data Types

Chapter 9

9.1 Numeric variables

📖 9.1.1

Variables of numeric type (**int** and **float**) are able to store values of numeric type and execute the following operations with them:

- + (sum) *a + b*, e.g.: *10 + 3 = 13*
- - (difference) *a – b*, e.g.: *10 - 3 = 7*
- * (product) *a * b*, e.g.: *10 * 3 = 30*
- / (quotient) *a / b*, e.g.: *10 / 4 = 2.5*
- // (integer quotient) *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*, therefore *5 * 5 * 5*

📝 9.1.2

Fill in the correct results for each operation:

```
100 + 133 = _____
125 - 37 = _____
12 * 13 = _____
15 / 4 = _____
100 // 3 = _____
18 % 5 = _____
1 ** 3 = _____
```

- 0
- 166
- 1
- 333
- 1
- 4
- 3.75
- 233
- 33
- 156
- 30
- 4.25
- 3
- 323
- 3
- 88

9.1.3

What is the result of the following arithmetic expression?

```
2**3 // (10 - 6)
```

9.1.4

Integers have a special operation that returns a remainder when divided. The % operator is used to calculate it.

E.g.:

```
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
```

In the last case, there was an error - you cannot divide by zero...

9.1.5

What is stored in the variable x after the program ends?

```
y = 15
z = 4
x = (y // z) ** (y % z)
```

9.1.6

Through the modulo operation (%), we can determine whether the entered number is even or odd. It is valid that for odd numbers the remainder after division by two is 1, for even numbers it is 0.

The program that provides us with this test can take the form of:

```
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.')
```

In the programs, we just changed the condition and swapped the contents of the positive and negative branches.

9.1.7

Fill in the correct results for each operation:

```
100 % 133 = _____
125 % 20 = _____
12 % 6 = _____
15 // 4 = _____
10 // 4 = _____
18 % 2 = _____
1 % 3 = _____
```

- 0
- 3
- 0
- 1
- 4
- 6
- 100
- 3
- 0
- 4
- 5
- 2
- 133
- 2
- 5
- 2
- 1

9.1.8

When working with integers, we must not forget negative numbers, which represent exactly half of all recordable values. We write a negative number using the - sign before the numerical value.

E.g.:

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

If we want to stay true to mathematical notation, we can enclose a negative value in parentheses, e.g.:

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

9.1.9

What will be stored in the variable p after the completion of the following steps of the program?

```
a = -3
b = 15 // -5
p = a - b
```

9.2 Abbreviated entry

9.2.1

In addition to the basic operators, **Python** has a special abbreviated 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. For example the entry

```
poc = poc + 1
```

can be shortened to the form

```
poc += 1
```

The abbreviated entry could be read as increasing the value of the variable **poc** by one.

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

Similarly, the entry

```
poc = poc - 3
```

is equivalent to the entry

```
poc -= 3
```

We change the value of the variable **poc** by subtracting the value 3 from it.

9.2.2

What will be stored in the poc variable after the execution of the following steps:

```
poc = 10
poc += 7
poc += 4
poc -= 5
```

9.2.3

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

```
x = x  * 10
x *= 10
```

Increases the value in variable **x** by a factor of ten.

```
b = b  - 15
b -= 15
```

Decreases **b** by 15.

Such abbreviated operators can be used in combination with all basic arithmetic operators.

=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

```
amount = amount / 2
amount /= 2
```

Halves the value stored in the **amount** variable.

```
amount = amount // 3
amount //= 3
```

Executes an integer division of the contents of the **amount** variable by three.

```
number = number ** 4
number **= 4
```

It powers the contents of the **number** variable to the fourth.

9.2.4

What will be stored in the poc variable after the execution of the following steps:

```
poc = 10
poc *= 7
poc //= 4
poc -= 5
poc //= 6
poc **= 3
```

9.2.5

How would we write the following assignment in abbreviated form?

```
p = p % 10
```

9.2.6

What notation can be used as equivalent to the given command?

```
v = v * 8
```

- v *= 8
- v ** 8
- v =* 8
- v * 8

9.2.7

Complete the corresponding symbols so that the entry is correct.

```
a _____ 3 # a = a / 3
b _____ 2 # b = b % 2
```

9.2.8

In the abbreviated entry, not only the numerical value may appear on the right-hand side, but we can also use a variable just as well.

The entry

```
a += b
```

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

Program

```
a = 10
b = 5
a *= b
print(a)
```

prints the value 50.

9.2.9

What will be the result of the following program?

```
a = 2
a += 3
a *= 2
b = 20 % a
b -=2
b += a
print(b)
```

9.3 Integers (programs)

9.3.1 Integer division

Write a program that divides two integers and determines the quotient and remainder (use operations for integer division). Treat division by zero at the beginning of the program and if it happens, print: "Cannot divide by zero". Otherwise, print the quotient and the remainder separated by a space.

```
Input:
4
5
Output:
0
4
```

```
Input:
9
```

```
0
Output: Cannot divide by zero
```

⌨ 9.3.2 Triangle

Write a program that checks whether the three entered numbers can be the lengths of the sides of a triangle. The input contains three integers. If these values can be the sides of a triangle, print the perimeter of that triangle. Otherwise, print -1.

```
input : 3
4
5
output: 12
```

```
input : 1
2
3
output: -1
```

⌨ 9.3.3 Even numbers

Write a program that prints even numbers from 1 to n.

```
Input: 20
Output:
2
4
6
8
10
12
14
16
18
20
```

⌨ 9.3.4 Prime number

Write code that checks whether the given number greater than 2 is prime. An integer greater than 2 is entered at the input. If the number is prime, print yes, otherwise print no.

```
input : 5
output: yes
```

```
input : 9
```

```
output: no
```

```
input : 11
output: yes
```

⌨ 9.3.5 The number of divisors of the entered number

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

```
Input : 7
Output: 2
```

```
Input : 12
Output: 6
```

```
Input : 100
Output: 9
```

⌨ 9.3.6 Perfect number

In number theory, a perfect number is a positive integer equal to the sum of its own positive divisors, i.e. sum of positive divisors without the number itself. Write a program that checks whether an entered number is perfect. The input contains a positive integer.

If the given number is perfect, print the value yes, otherwise, print the value no.

```
input : 28
output: yes
```

```
input : 999
output: no
```

⌨ 9.3.7 Coins + banknotes

Write a program that, for a given integer amount in euros, prints the minimum number of banknotes and the minimum number of coins that can be used to pay for this amount, as well as their breakdown in order from the largest to smallest. Consider also banknotes with a 500 euros value.

```
input: 3
output:
banknotes: 0
coins: 2
breakdown:
1 x 2
```

```
1 x 1
```

```
input: 13
output:
banknotes: 1
coins: 2
breakdown:
1 x 10
1 x 2
1 x 1
```

Decimal Numbers

Chapter 10

10.1 Decimal numbers

📖 10.1.1

Many task can be solved using integer operations, but there are also problems in which we need to use decimal numbers. To store decimal (real) numbers, Python uses the **float** data type.

The decimal part is **separated** from the integer **by a dot**.

```
5.18
```

We can recognize the real number in the output by the fact that its **decimal part is displayed**. A number with a zero decimal part has it displayed as .0, e.g.

```
14.0
```

We inform the compiler that the variable is supposed to be of **float** type by inserting a decimal value.

```
a = 12.5
b = 3.0    # we enter integer as a decimal number
```

📝 10.1.2

Choose the correctly written decimal numbers:

- 3.1
- 15.8
- 17.0
- 0.59
- 17
- 0,58
- 2,5

📖 10.1.3

Real numbers are written in a standard format:

```
3.1415296536, 556.44
```

or in scientific format:

```
5.5644e2
```

which means *5,5644 * 10^2 = 556,44*.

10.1.4

Find the correct decimal number for numbers written in scientific format:

```
1.1234e2 = _____
2.4532e3 = _____
2.4532e4 = _____
2.4532e1 = _____
5.048e0  = _____
3.2e-1   = _____
1.2e-3   = _____
```

- 1123.4
- 0.5048
- 0.012
- 11.234
- 2453.2
- 0.32
- 24.532
- 112.34
- 24532.0
- 5.048
- 0.032
- 0.0012
- 50.48
- 0.12
- 3.2
- 245.32

10.1.5

Find the correct number written in scientific format for decimal numbers:

```
11.11  = _____
2.48   = _____
255.32 = _____
2553.2 = _____
0.12   = _____
```

- 2.5532e2
- 1.111e1
- 2.5532e4
- 1.111e2
- 1.2e0
- 2.5532e1
- 2.5532e2
- 2.48e0

- 2.5532e3
- 1.2e1
- 1.2e-1
- 1.111e3
- 2.48e1

📖 10.1.6

When an integer and a real number type or operation for real numbers are combined in a calculation, the result is a real number (a number with a decimal point).

The result of the following code:

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

is a real number. This fact is presented by printing the result in the form

```
2.0
```

📝 10.1.7

What is the result of the activity of the following code?

```
a = 2
b = 3
c = b // a * b / a
print(c)
```

- 1.5
- 1,5
- 1
- 2.25
- 2,25

📝 10.1.8

What does the following program print?

```
a = 2.0
b = 3
c = a * b
print(c)
```

📖 10.1.9

To load a decimal number from the user, we use **input()** and then transform the text into a decimal number using the **float()** function:

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

📝 10.1.10

Complete the code for the addition of two decimal numbers:

```
aa = input('Enter the 1st number: ')
a = _____(aa)
bb = _____('Enter the 2nd number: ')
b = _____(bb)
print(a + b)
```

10.2 Functions for working with numbers

📖 10.2.1

We already know that Python can work with values of different types. Each data type has a set of standard functions with which we can process the values. We call them **built-in functions** because we don't need to add any plugins to the program to use them.

The function processes the value or values that we enter in the brackets (we call them parameters or arguments) and **returns a result** that we can work with further, e.g. print it or put it into a variable.

Each data type has its own built-in functions. For example, numeric data types have the following functions:

- **abs()** - returns the absolute value of the entered number,
- **max()** - returns the maximum value from the entered values,
- **min()** - returns the minimum value from the entered values,
- **pow(x, y)** - returns x to the power of y, this is the same calculation as x ** y

```
abs(-3)                 # result 3
max(2, 5, 6, 8, 1, 3)   # result 8
min(2, 5, 6, 8, 1, 3)   # result 1
pow(3, 2)               # result 9
```

10.2.2

Complete the function so that the result is correct:

```
a = -8
print(_____(a)) # prints 8
```

10.2.3

Complete the function so that the result is correct:

```
a = 7
b = 2
print(_____(a, b)) # prints 2
```

10.2.4

In addition, the float data type has a **round()** function that processes a decimal value by rounding it to an integer value.

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

10.2.5

Fill in the correct rounding results:

```
a = round(6.8) # result _____
b = round(13.6) # result _____
c = round(3.12) # result _____
d = round(4.51) # result _____
```

10.2.6

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

- down if .5 is preceded by an even number, e.g.

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

- up if .5 is preceded by an odd number, e.g.

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

10.2.7

Fill in the correct rounding results:

```
a = round(6.5) # result _____
b = round(13.5) # result _____
c = round(3.5) # result _____
d = round(4.5) # result _____
```

- 13
- 14
- 5
- 4
- 3
- 4
- 7
- 6

10.2.8

What is the result of the following program?

```
a = 10
b = round(a/3*100)/100
c = a - b
d = round(c)
print(d)
```

10.2.9

For rounding to a specified number of decimal places, an extended version of the **round()** function is used, where the second parameter specifies the number of decimal places to which the value is supposed to be rounded.

For example, to round to two decimal places:

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

The result is 3.14

If a negative value is entered, the value is rounded to tens, hundreds, etc. For example:

```
round(1234, -1)  # result 1230
round(1234, -2)  # result 1200
round(1254, -2)  # result 1300
```

```
round(2854, -3)  # result 3000
```

10.2.10

Fill in the correct results:

```
round(13.67, 1)   # result _____
round(1.865, 2)   # result _____
round(136, -1)    # result _____
round(387, -2)    # result _____
round(3254, -2)   # result _____
round(8154, -3)   # result _____
```

- 3300
- 400
- 390
- 3250
- 8200
- 1.87
- 1.86
- 3260
- 14
- 13.6
- 8000
- 140
- 13.7
- 1.90
- 130

10.3 Nesting functions

10.3.1

Despite the fact that we have already used this approach quite naturally several times, we will explain how functions can be nested within each other.

As we said, each function processes arguments and returns a result that we can use further, e.g.

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

We can combine this entry into a single command by omitting the assignment of the result of the **max()** function to a variable and simply printing the result:

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

The compiler evaluates this notation by first calculating the maximum of 10 and 20, returning the result, and using it as an argument for the print command.

We can also use:

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

where the **max()** function is evaluated first and its result subsequently becomes the argument of the round() function. Then the result is assigned to the variable **a**.

In this way, we can nest practically any number of functions into each other.

10.3.2

What is the result of the following sequence of commands?

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

10.3.3

What is the result of the following sequence of functions?

```
print(round(min(max(3.1, 5, 4.6, 7.8), min(10.8, 15.62, 3.21,
11))))
```

10.3.4

Just as mathematical functions, all other functions can also be nested and processed.

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

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

However, this procedure can be simplified thanks to nesting the **input()** function in the **int()** function that transforms text into a number:

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

The process is the same with decimal numbers:

```
b = float(input())
```

10.3.5

Complete the code to get inputs:

```
x = _____(_____())     # loading an integer
y = _____(_____())   # loading an decimal number
```

- get
- float
- input
- integer
- input
- str
- str
- int
- get

10.4 Decimal numbers (programs)

10.4.1 The sum of two real numbers

Write a program that, for two real numbers, calculates their sum, rounds it to a whole number and prints it.

```
input:
8.2
2.5
output: 11
```

```
input:
1.5
3.3
output: 5
```

10.4.2 Seconds

Write a program that reads an integer representing the number of seconds from the input. Write how many days, hours and minutes it represents. Round the results to 3 decimal places.

```
Input: 3600
Output:
```

```
days: 0.042
hours: 1.0
minutes: 60.0
```

```
Input: 500000
Output:
days: 5.787
hours: 138.899
minutes: 8333.333
```

⌨ 10.4.3 BMI calculation

Write a program that, based on the entered weight and height in meters, calculates the BMI index and prints whether you are overweight or not.

BMI (body mass index) is calculated as the ratio of weight in kilograms to the square of height in meters.

- BMI < 18,5 - underweight,
- 18,5 <= BMI < 25 - normal weight,
- 25 <= BMI < 30 - overweight,
- BMI > 30 - obesity.

```
Input :
45
1.70
Output: underweight
```

```
Input :
90
1.65
Output: obesity
```

```
Input :
80
1.80
Output: normal weight
```

⌨ 10.4.4 Circle

Write a program that, for an entered radius (decimal number), calculates and prints the area and circumference of a circle rounded to whole numbers. Let the variable **pi** have the value 3.14.

```
input : 5
output: 79 31
```

```
input : 4.5
output: 64 28
```

⌨ 10.4.5 Time to march

Write a program that reads from the input the number of kilometers to the destination and the expected speed of the tourists. Then it will print how many hours the march will take.

```
Input: 21
6
Output: 3.5
```

```
Input: 100.5
10
Output: 10.05
```

⌨ 10.4.6 Arrival

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

Then the program will display how many kilometers the fuel in the tank is sufficient for.

```
Input: 20
8
Output: 250
```

```
Input: 50
5
Output: 1000
```

⌨ 10.4.7 Rate transfer

Write a program that converts an entered number of EUR to USD. The first value will be the amount in EUR, the second the conversion rate. The output will be the number of USD that can be obtained for the entered number of EUR

```
Input: 100
1.1
Output: 110.0 $
```

```
Input: 1000
```

```
1.008
Output:1008.0 $
```

Boolean Expressions

Chapter 11

11.1 Boolean expression

📖 11.1.1

In addition to expressions that result in a number, we often work with expressions that result in true or false. We refer to these expressions as **boolean** and we have already encountered them when using conditions in the **if** statement.

Although so far we have only compared variables with each other or variables with values, we can also compare values themselves.

An example can be an entry

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

which can be replaced directly with a command

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

If the expression in parentheses is true, the result is **True**, otherwise it is **False**.

Eg: is it true that 5 - 3 < 0? It's not.

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

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

📝 11.1.2

Complete the code that tests whether **n** is positive.

```
n = 28
print(n _____)  # The output is the value _____
```

- > 50
- > 0
- false
- True
- <| 0
- true
- False

11.1.3

Complete the code that tests whether twice **n** is less than 50.

```
n = 28
print(2 * n _____) # The result is value _____
```

- true
- <| 0
- False
- > 50
- True
- < 50
- false

11.1.4

If we can print the evaluation of an expression, we can also insert it into a variable:

```
result = 4 > 0
```

In the code, it is first evaluated whether 4 > 0 and then the **True** value corresponding to the truth is inserted into the variable **result**.

In this code, it is first evaluated whether 4 > a + 5, i.e. 15, and then the **False** value corresponding to false is inserted into the variable **result**.

```
a = 10
result = 4 > a + 5
```

11.1.5

What is the result of the program?

```
a = 7
t = a < 5
print(t)
```

- False
- True

11.1.6

To store boolean values, we use variables of type boolean (bool). We usually get their content as a result of comparison, verification of the truth of the condition, etc.

We can verify the condition, whether **a > b**, by writing it in the **if** structure:

```
if a > b
```

but we can also store the result of the expression evaluation in a variable

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

If the value of **a** is greater than **b**, the value of **True** is stored in the variable **result**, otherwise (less than or equal to), the result variable will contain the value of **False** after evaluation.

We can also check the data type of the resulting value.

```
print(type(result)) # prints <class 'bool'>
```

11.1.7

Check whether the variable **a** contains a value of **5** and store the result in the variable **t**, whose value wiil be printed.

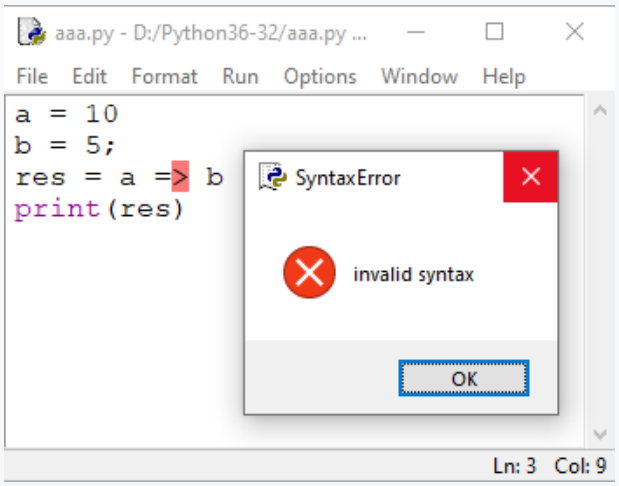
```
a = 7
t = a _____ 5
print(t)
```

11.1.8

Comparison operators are attached to the boolean type, we will repeat them so we are complete:

- **>** - is greater than, e.g. **a > b**
- **>=** - is greater or equal to, e.g. **a >= b**
- **<** - is lesser than, e.g. **a < b**
- **<=** - is lesser or equal to, e.g. **a <= b**
- **==** - is equal to, e.g. **a == b**
- **!=** - is not equal, e.g. **a != b**

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



11.1.9

Which comparison operators are correct?

- >=
- <|=
- ==
- !=
- <|>
- =>
- =<|

11.2 Using expressions

11.2.1

The result of the comparison can be also used in conditions by first finding the result of the expression and then using it in the condition, e.g.:

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

Although such a procedure is not standard for simple conditions, it will help us understand the principle of using boolean expressions.

11.2.2

What does the following program print?

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

- ends with an error
- values are equal
- values are different

11.2.3

The entry

```
if result == True
```

we usually write in the form

```
if result
```

because the evaluation of the condition **result == True** depends on what value the variable **result** has.

If it is true,

```
if result == True
```

we ask if true is true (**True == True**) - the result is **True**.

If the variable contains a false value:

```
if result == True
```

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

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

- if it contains a true value, the result is **True**,
- if false, the result is **False**.

11.2.4

Complete the code so that it prints whether it is a negative or non-negative number (allow to enter decimal numbers as well).

```
input_ = input('Enter a number: ')
a = _____(input_)
negative = a _____ 0;
if _____:
    print("negative")
else:
    print("non-negative")
```

- negative
- <
- float
- int
- negative==False
- negative==0
- negative == True

11.2.5

Complete the program to verify whether **n** is even.

```
n = 23
print(n % 2 _____ 0)
```

The output value is

```
_____
```

11.2.6

Complete the program to check whether **n** is non-zero.

```
n = -6
print(n _____ 0)
```

The output value is

```
_____
```

11.3 Compound conditions

📖 11.3.1

In the program, we often combine several conditions that can be in different relationships. We most often encounter situations in which:

- all conditions must apply,
- it is enough if only one of the conditions applies.

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

The task can be solved as follows:

```
age = int(input('Enter the age of an employee: '))
if age >= 18:        # whether the first condition is met
  # check whether the age is simultaneously less than the
upper limit
  if age <= 70:      # both conditions are met
    print("this employee is in productive age")
```

📝 11.3.2

Arrange the rows of the program, so that it prints the season of the year that the entered moth belongs to. Arrange the rows so that they are verified in the order of spring, summer, autumn, winter.

- if month >= 7:
- print("winter")
- if month <= 12:
- print("summer")
- if month <= 6:
- if month >= 4:
- print("spring")
- if month >= 10:
- print("autumn")
- if month >= 1:
- if month <= 3:
- if month <= 9:
- month = int(input('Enter the month: '))

📖 11.3.3

Original program:

```
age = int(input('Enter the age of an employee: '))
```

```
if age >= 18:
  if age <= 70:
    print("this employee is in productive age")
```

will be slightly modified.

A simpler entry allows us to write two verifications into a single **compound condition**. The fact that they should apply simultaneously is expressed through the boolean conjunction **and** (and simultaneously). This is how we simplify the entry of two conditional commands by combining them into one compound condition:

```
age = int(input('Enter the age of an employee: '))
if (age >= 18) and (age <= 70):
    print("this employee is in productive age")
```

In a compound condition, complete conditions are combined - i.e. **variables must be specified in each subcondition.**

> **Python** does not require the use of parentheses when creating compound conditions, but we recommend them to avoid various mistakes.

11.3.4

Let´s also modify our second program - complete the program so that it prints the season of the year for the entered month.

```
month = int(input('Enter the month: '))
if month >= 4 _____ month <= 6:
     print("spring")
if month >= 7 _____ month <= 9:
     print("summer")
if month >= 10 _____ month <= 12:
     print("autumn")
if month >= 1 _____ month <= 3:
     print("winter")
```

- and if
- if
- if
- and
- and if
- and
- and if
- and
- and
- if

- if
- and if

11.3.5

A specific feature of the **Python** language is the possibility of delimiting a variable with comparison operators from both sides, which allows us to significantly shorten the entry of boolean expressions.

> Attention, we only do this if the same variable appears in both conditions, therefore it is not a solution that can be used anytime and anywhere.

The expression

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

can be written as

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

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

```
age = int(input('Enter the age of an employee: '))
if (18 <= age <= 70):
    print("this employee is in productive age")
```

or

```
if (70 >= age >= 18):
    print("this employee is in productive age")
```

11.3.6

Arrange the lines of the program which, based on the diameter of the egg given in millimeters, evaluates the size category the egg belongs to. Verify the average value in from largest to smallest order.

- elif 65 < p <= 55:
- print('L')
- else:
- if p >= 65:
- print('XL')
- elif 55 < p <= 45:
- print('S')
- print('M')
- p = int(input('Enter the diameter of the egg in millimeters: '))

11.3.7

Adjust the program for the seasons again - complete the program so that it prints the season that entered month belongs to.

```
month = int(input('Enter the month: '))
if 4 _____ month _____ 6:
     print("spring")
if 7 _____ month _____ 9:
     print("summer")
if 10 _____ month _____ 12:
     print("autumn")
if 1 _____ month _____ 3:
     print("winter")
```

- <=
- <=
- >=
- <=
- <=
- <=
- >=
- >=
- <=
- <=
- >=
- >=
- >=
- >=
- >=
- <=

11.3.8

Complete the boolean expression so that it tests whether the value of the variable **n** belongs to the interval **<-5, 5>.**

```
n = 0
print(-5 _____ n _____ 5}
```

The result is

```
_____
```

11.3.9

In some cases, we require only one of the verified conditions to be met. In such case, the boolean conjunction **or** is used.

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

The evaluation of the expression is true if at least one of the conditions is met - i.e. if **a > 0** or **b < 0**.

If both conditions are met, the expression is also true.

11.3.10

Complete the program so that it prints whether the applicant is entitled to an allowance, the allowances are intended for persons under 18 and over 70.

```
age = int(input("Enter the age: "))
if (age _____ 18) _____ (age > _____):
    print("the person is entitled to an allowance")
```

- <
- 70
- 18
- or
- >
- and

11.3.11

Complete the program so that it prints that a number is accepted if it is positive or even.

```
number = int(input('Enter a number: '))
if (number > _____) _____ (number _____ 2 _____ 0):
    print('accepted')
```

- or
- //
- %
- >
- and
- 18
- ==
- <|
- /
- 0

- **

11.4 Evaluation of the compound expressions

📖 11.4.1

We achieve the simultaneous validity of several conditions by using the conjunction **and.** In case it is sufficient for us to fulfill only one condition from the group listed, we use the conjunction **or**.

The use of this pair is not limited to use in the conditional **if** statement - we can also use them when working with boolean expressions.

For example the result of the expression stored in the variable **c**

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

can be obtained by gradually evaluating the individual parts of the compound condition. First, we evaluate each part separately:

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

the result of a combination of truth values **True or False - True.**

📝 11.4.2

What is the result of the following program?

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

- False
- True

📖 11.4.3

Let's test the calculation for the requirement that the number be both positive and even.

```
n = 15
```

Let's test whether the number is positive and even.

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

we connect with the boolean conjunction and

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

11.4.4

What is the result of the following program?

```
a = 1
b = 5
c = a >= b and b > 0
print(c)
```

- False
- True

11.4.5

In addition to checking whether the condition is true, it is sometimes convenient to use the entry: if it is not true, then e.g.:

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

An entry beginning with the expression **not negates** the result of the expression or the content of the variable listed after it - it will make the value **True False** and vice versa.

In this case, the **zeroDivisor** variable contains the value **False** and the entry in the condition means:

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

11.4.6

What command do we use to negate the contents of a boolean variable?

- not
- nor
- or
- xor
- no
- now

11.4.7

The combination of logical expressions and logical variables need not be limited to only two elements. The evaluation proceeds by first evaluating the expressions in parentheses, then the negation, and then proceeds from left to right.

E.g.

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

11.4.8

What is the resulting value of the following expression?

```
a = 1
b = 2
result = (a == 5) and (b < 6) and (a > b)
```

- False
- True

11.4.9

What is the resulting value of the following expression?

```
a = 2
b = 2
result = (a == b) and (b > 6) or (2*a > 2*b)
```

- False
- True

11.4.10

What is the resulting value of the following expression?

```
k = 2
j = 3
result = (k <= 5) or (j > 6) and (j >= k)
```

- True
- False

11.4.11

What is the resulting value of the following expression?

```
a = -2
b = 2
result = (a != b) or (b > 6) and not (a > -a)
```

- True
- False

11.4.12

What is the resulting value of the following expression?

```
k = 5
j = 6
result = ((k <= 5) and (j > 6)) or not(k > j)
```

- True
- False

11.5 Boolean expressions (programs)

⌨ 11.5.1 Bigger/smaller

Write a program to find out whether the first of two given numbers is less than the second.

```
input:
2
4
output: True
```

```
input:
5
2
output: False
```

⌨ 11.5.2 Report card

For the entered average on the report card, write whether the student:

- **passed with honors** - average less than or equal to 1.5;
- **did very well** - average greater than 1.5 and less than or equal to 2;
- **passed** - average greater than 2 and less than or equal to 4;
- **failed** - average more than 4.

Use a dot as a decimal separator (1.3, 2.8, etc.)

```
Input: 1.5
Output: passed with honors
```

```
Input: 4.1
Output: failed
```

⌨ 11.5.3 Maximum of three numbers

Write a program that prints the largest of the three entered numbers. If all three numbers are equal, it will print "Numbers are equal".

```
Input: 2 4 6
Output: 6
```

```
Input: 2 2 2
Output: Numbers are equal
```

⌨ 11.5.4 Multiples

Write a program that, for three entered numbers, determines whether any of them is a multiple of two others. If so, it prints the given number, otherwise it prints **False**.

```
Input:
3
2
6
Output: 6
```

```
Input:
5
2
6
Output: False
```

⌨ 11.5.5 Interval

Write a program that checks whether the entered number is in the entered interval. At the beginning of the algorithm, check whether the entered interval is correctly rotated (e.g. not 5.2 but 2.5) and if not, adjust it.

The input contains a trio of integer values representing two interval limits and the entered value.

The output will be a correctly rotated interval, and information on whether the entered number belongs to the interval.

```
Input : 5
10
7
Output: <5,10> 7 belongs
```

```
Input : 100
20
10
Output: <20,100> 10 does not belong
```

```
Input : 30
4
85
Output: <4,30> 85 does not belong
```

⌨ 11.5.6 Test rating

Write a program that, after entering the percentage of success in the test, prints a verbal rating according to the following rules

- More than 90 percent: **Excellent performance**.
- More than 70 percent, or lees than or equal to 90 percent: **Great performance**.
- More than 50 percent, or lees than or equal to 70 percent: **Good job**.
- More than 30 percent, or lees than or equal to 50 percent: **Not worst, but you can do better**.
- Less than or equal to 30 percent: **You need to work on yourself. Next time it will be better.**

E.g.:

```
Input: 65
Output: Good job.
```

Strings

Chapter 12

12.1 String data type

📖 12.1.1

A string is a data type that allows you to store and work with a group of characters that usually make up a word or continuous text.

In the program, string is delimited by apostrophes (') or quotation marks (").
Because of them, the compiler knows how to work with the given value.

E.g.:

```
name = 'Adam'
```

insert the text content Adam into the **name** variable

For the entry

```
name = Adam
```

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

📝 12.1.2

Which characters delimit a text string in Python?

- '
- "
- !
- #
- ()

📖 12.1.3

We already know that the simplest operation that can be executed on strings is to concatenate them. This operation is provided by the "+" character, which from two existing strings creates a new one by appending the contents of the second to the contents of the first string.

```
output = 'it' + 'bites'
print(output) # prints itbites
```

We can concatenate any number of strings or variables that contain the string.

```
a = 'Mama'
b = ' has '
```

```
c = 'Ema'
d = a + b + c
print(d) # print Mama has Ema
```

12.1.4

What will be stored in the variable **d** after the following program is executed?

```
a = "Warning"
b = "dark"
c = "!"
d = a + b + c
```

- Warningdark!
- warning dark
- Warning dark!
- Warning dark !
- WarningDark!

12.1.5

To concatenate a string and a number, we need to use the conversion of a number to a string using the **str()** command.

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

In case we would like to use the "+" sign to combine text and number, e.g.

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

we get an error message:

```
TypeError: can only concatenate str (not "int") to str
```

12.1.6

Complete the program so that you get an output in the form

```
a + b = c
```

e.g.

```
10 + 20 = 30
```

```
a = 3              # number
b = 7              # number
c = a + b          # number
d = _____(a) + '_____' + _____(b) + ' = ' + _____(c)
print(d)
```

📖 12.1.7

The variable type containing a text string is referred to as string.

The command **type(variable)** returns the value of **str**:

```
a = "Python”
print(type(a))
```

prints

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

📝 12.1.8

Assign the correct types to the variables:

```
a = 3
b = 7.5
c = "Prague"
d = a == b
print(type(a))   # _____
print(type(b))   # _____
print(type(c))   # _____
print(type(d))   # _____
```

- Bool
- double
- bool
- integer
- Str
- int
- bool
- str
- str

- int
- float

📖 12.1.9

We also remember that when data is loaded, they all enter the program as text strings, and for further processing in a different way, it is necessary to convert them to appropriate values.

```
print('Enter an integer: ');
input1 = input()
a = int(input1)                          # convert to an integer
print('Enter a decimal number: ');
input2 = input()
b = float(input2)                        # convert to a decimal
number
c = a - b
result = 'Difference: ' + str(c)      # convert the number to
text
print(result)
```

📝 12.1.10

Complete the program so that it prints the product of two decimal numbers in a clear form. For example for inputs: 1.1 and 2.2 will print

```
1.1   * 2.2 = 2.42
```

```
print('Enter 1st decimal number: ');
v1 = input()
a = _____(v1)
print('Enter 2nd decimal number: ');
v2 = input()
b = _____(v2)
c = a _____ b
result = _____(a) + ' * ' + _____(b) + ' = ' + _____(c)
print(result)
```

12.2 String multiplication

📖 12.2.1

In addition to the sum operation, Python also allows you to use multiplication when working with strings.

When multiplying, one variable must be of type string and the other of type integer, e.g.

```
n = 3
txt = 'uff '
c = n * txt
```

The result of the multiplication is a text string containing the contents of the text variable repeated n times in a row.

In this case it is

```
uff uff uff
```

> Other operations such as division or subtraction are not used on strings.

📝 12.2.2

What is the result of the following command?

```
print('ab' * 3)
```

- ababab
- ab3
- 3ab
- aaabbb
- error

📝 12.2.3

What is the result of the following command?

```
print('ab' + 3)
```

- error
- ab3
- 3ab
- aaabbb
- ababab

12.2.4

What is the result of the following command?

```
print(str(5) * 3)
```

- 555
- 15
- 535353
- 5553
- error

12.2.5

What will be the result of the following command?

```
print(str(5) + 3)
```

- error
- 53
- 5 3
- '5'3
- 5553

12.2.6

What is the result of the following program?

```
ret = '101'
ret *= 10
print(ret)
```

12.2.7

Which of the following commands is possible?

```
x = input()
```

- print(x + x)
- print(x * x)
- print(x - x)

12.3 Characters in string

📖 12.3.1

Every variable of the string type allows searching, finding the number of characters, changing the size of characters, etc.

The simplest operation is to return the number of characters in the stored content. We get it through the **len()** command.

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

The number of characters contained in the data variable is stored in the **length** variable, i.e. in this case **4**.

📝 12.3.2

What will be stored in the **length** variable after the following code is executed?

```
x = 'python'
length = len(x)
```

📝 12.3.3

What does the following code print?

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

📖 12.3.4

Every string consists of characters. Each character has its place in the string, which is defined by an index. Python counts the elements in any list by starting to count from zero.

The first character in the string is at position 0, the second at position 1, and so on. The last character is at a position one less than the total number of characters in the string.

E.g. for:

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

characters are distributed in individual positions as follows:

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

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

12.3.5

Which character is in position 1 in the string?

```
ret = 'Priscilla'
```

12.3.6

Access to characters at individual positions is provided by notation consisting of the name of the variable followed by square brackets with an index, e.g.:

```
data = 'Python'
first = data[0]
```

The outputs for the following commands then are:

```
data = 'Python'
first = data[0]
print(first)   # prints P
print(data[1]) # prints y
print(data[2]) # prints t
print(data[3]) # prints h
print(data[4]) # prints o
print(data[5]) # prints n
print(data[6]) # prints an error
```

Notice that the last character of the string has an index one unit smaller than the length of the entire string, i.e.

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

12.3.7

Which character does the following code print?

```
ret = 'Priscilla'
print(ret[4])
```

12.3.8

Fill in the universal correct index of the last character.

```
ret = input()
print(ret_____________________________________________)
```

- [
- 2
- [
-)
- 1
- }
- (
-]
- -
- len
- {
- +
- *
- (
-)
-]
- ret

12.3.9

If we try to access a character that does not exist,

```
data = 'Test'
char = data[4]
```

an error message is displayed.

```
IndexError: string index out of range
```

It informs us that we are out of range of the string indexes.

This is a fairly common mistake of a novice programmer.

12.3.10

Which calls can be used to get a character from the string ret?

```
ret = 'Anaconda'
```

- ret[0]

- ret[7]
- ret[len(ret)-1]
- ret[4]
- ret[8]
- ret[9]
- ret[len(ret)]

12.4 Characters iteration

🕮 12.4.1

We can access each character in the string through its index. If we need to go through all the characters, we usually do so by iterating using the loop.

Let's write the entered word one letter at a time.

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

The **range(length)** command will generate values starting with zero and ending with **length-1,** which is exactly the index of the last character we need.

12.4.2

Complete the program that prints all the characters in the string below each other:

```
ret = input('Enter a string: ')
length = _____(ret)
for i in _____(_____)_____
    print(_____________________)
```

- range
-)
-]
- i
- :
- lenght
- [
-)
- ;
- ret
- Range
- (

- [
- length
-]
- {
- in
- len
- }
- (

📖 12.4.3

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

The procedure is very simple, just adjust the parameters of the previous cycle.

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

📝 12.4.4

Complete the program that prints letters in odd positions in the entered text (characters in position 1, 3, 5, ...).

```
ret = input('Enter a string: ')
length = _____(ret)
for i in range(_____, _____, _____):
  print(ret[i])
```

📖 12.4.5

In Python, 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)
```

The loop then works in such a way that at each step of the loop the next character in the specified string is inserted into the variable i.

The output of the program will be:

```
S
l
```

```
o
v
a
k
i
a
```

We can use this iteration if we want to process only the characters of the string, but we are not interested in their position.

12.4.6

Complete the code so that we reach the listed output.

```
ret = _____

for i in _____:
  print(i)
```

```
P
o
k
e
m
o
n
```

12.5 Typical tasks

12.5.1

Write a program that detects how many times the character a is found in the entered word.

Complete the program that will solve the task:

```
word = _____('Enter a word: ')  # read the word from the input
length = _____(word)              # find the number of
characters
#in the variable number will be the number of characters found
and - 0 at the beginning
number = 0
for i in _____(length):              # in the loop we iterate
through the positions from 0 to length-1
```

```
   if word[i] _____ 'a':                # if there is an 'a' in
position i, we increase the number
       number = _____ _____ 1
print('The word contains',_____,'characters _____a"') # output
```

12.5.2

We'll repeat the task again, using Python's ability to iterate through the characters of a word directly in a loop:

Write a program that detects how many times the character a is found in the entered word.

Complete the program that will solve the task:

```
word = _____('Enter a word: ')  # read the word from the input
number = 0                              # number of characters a
found - 0 at the beginning
for i in _____:                        # in the loopwe go through
the characters of the word
    if _____ == 'a':                      # if the examined
character is 'a', we increase the number
        number = _____ + 1
print('The word contains',number,'charatcters _____a"') #
output
```

12.5.3

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

Complete the program:

```
number = input('Enter a number: ')  # read the number from the
input
even = 0                           # the number of even and odd
numbers is 0
odd = 0
for i in _____:                     # we iterate trough the digits
   x = _____(i)                      # change each character
(digit) to a number
   if x _____ 2 _____ 0:                # if the digit is
divisible by 2 it is an even number
       even = even + 1
   else:                           # else odd
       odd = odd + 1
```

```
print('The number contains',even,'even and',odd,'odd digits.')
```

12.5.4

We will repeat the task again by using a compound condition. When reading the digits 0, 2, 4, 6, and 8, we increase the number of occurrences of even digits, otherwise the number of odd ones.

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

Complete the program:

```
number = input('Enter a number: ')  # read the number from the
input
even = 0                          # the number of both even and
odd numbers is zero
odd = 0
for i in _____:                    # in the loop we iterate
trough the digit of the word
    # if the examined character is 0,2,4,6,8 otherwise say 0
or 2 or...
    if i == '0' _____ i == '2' _____ i == '4' _____ i == '6'
_____ i == '8':
        even = even + 1           # increase the number of even
    else:                           # else odd
        odd = odd + 1
print('The number contains',even,'even and',odd,'odd digits.')
# output
```

- and
- and
- word
- or
- and
- or
- or
- or
- number
- and
- and

12.5.5

Python allows to replace a lengthy notation

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

which tests whether the variable has one or the other or the next value, using the **in** operator, which tests whether the value is in the list:

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

12.5.6

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

Complete the program:

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

12.5.7

Write a program that prints a mirror image of the entered word, e.g.:

```
Mama -> amaM
winter -> retniw
```

Although later we will also show commands that will simplify this activity, for now we will make do with a simple loop.

First, we will create a separate variable into which we will insert characters by inserting the next character in the sequence before the existing string, e.g. for the word Aladdin, we will proceed as follows:

- first we read **A** and store it in the result (result = "A")
- we read **l** and store it before the result obtained so far (result = "l" + result, i.e. "lA")
- we read a and store it before the result obtained so far (result = "a" + result, i.e. "alA"), etc.

```
word = input()
mirror = ''
```

```
for character in _____:
   # insert the evaluated character before the string obtained
so far
   mirror = _____ + _____
print(mirror)
```

12.6 Strings (programs)

⌨ 12.6.1 Occurrence of a digit

Write a program to find the number of times the digit '3' is present in the number entered at the input.

```
input : 3259873102
output: 2
```

```
input : 3333333333
output: 10
```

⌨ 12.6.2 Digit sum

Write a program that finds the digit sum of an entered number.

```
Input : 123
Output: 6
```

```
Input : 0124
Output: 7
```

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

⌨ 12.6.3 Number of digits

Write a program to find how many digits are in a string.

```
input : I have 2 slippers and I am 88 years old.
output: 3
```

```
input : 3333333333
output: 10
```

⌨ 12.6.4 Vowels

Write a program that finds out how many vowels the entered sentence contains.

Consider only characters without long and soft characters.

Consider both lower and upper-case letters.

```
input : Mama had Ema.
output: 5
```

```
input : WARNING, it is freezing.
output: 7
```

⌨ 12.6.5 Number of words

Write a program that finds out how many words are in the entered sentence.

```
input : Mama has ema.
output: 3
```

```
input : Winter started today: it started snowing in the
morning, it was gloomy at noon, and in the evening we built a
snowman.
output: 22
```

⌨ 12.6.6 Correction of a text

Write a program that changes all non-numeric characters in the specified string to the number 1 and prints the changed string.

```
Input : 57ada87
Output: 5711187
```

```
Input : 3.,úôéáá23Â§ô!3
Output: 31111111231113
```

⌨ 12.6.7 Decryption

Write a program that prints the decrypted text for an encrypted message.

You get the text by selecting every third letter from the input string. An encrypted message starts with the first letter.

eg. for **w**tz**a** i**r**nh**n**bh**i**hj**n**hj**g** i**d**p **o**gj**g**.

the result is **warning dog** - every third letter from the word

Characters and Special Outputs

Chapter 13

13.1 Characters in ASCII

📖 13.1.1

The basic building element of the string is the character.

We can compare characters based on their order, which resembles the alphabet

```
'a' < 'b' < 'c' … < 'z'.
```

However, it also applies that all uppercase letters are smaller than all lowercase letters

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

This behavior is a consequence of the computer's character encoding. Each character has its own numerical code, based on which the system knows what form to give the character.

Characters are usually represented by an ASCII table containing 255 basic characters. Although alphabets are currently encoded using Unicode/UTF-8 encoding, the first 128 characters are encoded the same way.

> Not all characters are displayable, only characters from 32 to 126 are used in printouts.

ASCII table:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
32	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
48		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
64	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
80	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
96	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
112	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
128	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

13.1.2

Choose the correct statements based on the position of the characters in the ASCII table:

- 'a'<|'z'
- 'A'<|'Z'
- 'Z'<|'a'
- 'a'<|'A'
- 'a'<|'Z'
- 'A'<|'1'

13.1.3

How do we find out whether the entered character is lowercase or uppercase?

If the character is present:

- between the first uppercase and the last uppercase character it is a capital letter,
- between the first lowercase and the last lowercase character it is a lowercase letter.

```
c = input('Enter a character: ')
if 'a' <= c <= 'z':
   print('lowercase letter')
elif 'A' <= c <= 'Z':
   print('uppercase letter')
else:
   print('it is not a letter')
```

13.1.4

Complete the code to determine whether the character in the variable c is a digit:

```
c = input('Enter character: ')
if '_____' <= c _____ '_____':
   print('it is a digit')
else:
   print('it is not a digit')
```

13.1.5

Thanks to the encoding of characters in the ASCII table, we can determine for each character its successor and predecessor. A couple of commands help us with this:

- **ord()** –returns the position of the character in the ASCII table, e.g. ord('A') returns 65
- **chr()** – returns the character that is at the specified position, e.g. chr(65) returns 'A'

13.1.6

Choose the correct command:

returning the position of the character in the ASCII table - _____, e.g. returns 66 for B

returning the character at the specified position - _____, e.g. for 66 it returns the value B

- order()
- chr()
- ord()
- getChar()
- inc()
- ascii()

13.1.7

For the entered character, print its predecessor and successor.

We will use the **ord()** and **chr()** commands:

- Pomocou **ord()** zistíme pozíciu znaku v ASCII tabuľke
- To get the predecessor, we use **chr()** to write a character at a position smaller by one.
- To get a successor, we use **chr()** to write a character at a position one larger.

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

13.1.8

What is the output of the command

```
print(chr(ord('A') + 1))
```

13.1.9

What is the output of the command

```
print(chr(ord('d') + 2))
```

13.1.10

What is the output of the command

```
print(chr(ord('7') - 2))
```

13.2 Comparison

13.2.1

Character positions are also used when comparing whole words (or more precisely strings).

We can determine the similarity of the strings through a simple comparison.

What does the following program print?

```
r1 = 'Mama'
r2 = 'Papa'
if r1 == r2:
  print('same')
else:
  print('different')
```

- different
- same
- program prints an error

13.2.2

However, string comparison tells us nothing about which string is alphabetically larger or smaller.

To determine the lexicographic (alphabetical) comparison, a "classic" comparison is used, which uses the positions of characters in the ASCII (Unicode) table.

It proceeds in both strings from the first position and when different characters are encountered, their position in the ASCII table is compared.

The string whose first distinct character has a lower position is smaller than the second.

For example for '**M**om' and '**D**ad' the very first character is different, and therefore:

```
'Dad' < 'Mom'
```

For 'Michal' and 'Michaela' the words differ only in the 6th character and 'l' > 'e', because

```
'Michal' > 'Michaela'
```

For 'Ivan' and 'Ivana' the first 4 characters are the same and the fifth character no longer exists in the word 'Ivan'. Since there is nothing less than the character 'a' in the last position in the word 'Ivana', then:

```
'Ivan' < 'Ivana'
```

13.2.3

Choose the correct statement

- 'parent' <| 'teacher'
- 'parent' > 'teacher'
- 'parent' = 'teacher'

13.2.4

Choose the correct statement

- 'Jasmina' > 'Aladin'
- 'Jasmina' <| 'Aladin'
- 'Jasmina' = 'Aladin'

13.2.5

Choose the correct statement

- 'spring' <| 'summer'
- 'spring' > 'summer'
- 'spring' = 'summer'

13.2.6

Choose the correct statement

- 'Daniel' <| 'Daniela'
- 'Daniel' > 'Daniela'
- 'Daniel' = 'Daniela'

13.2.7

Attention, in the case of 'summer' and 'Winter', it applies that

```
'summer' > 'Winter'
```

because lowercase letters are placed in higher positions than uppercase ones in the ASCII table.

13.2.8

Choose the correct statement

- 'aladin' > 'Jasmina'
- 'aladin' <| 'Jasmina'
- 'aladin' = 'Jasmina'

13.2.9

Choose the correct statement

- 'Jasmina' <| 'jasmin'
- 'Jasmina' > 'jasmin'
- 'Jasmina' = 'jasmin'

13.2.10

Choose the correct statement

- 'pear' > 'Pear'
- 'pear' <| 'Pear'
- 'pear' = 'Pear'

13.3 Numbers as a strings

13.3.1

Although we usually compare numbers based on mathematical rules, there may also be situations where we compare them lexicographically - like text.

Then it is true that

```
0 < 1 < 2… < 9
```

whereas we treat individual digits as characters.

Although it is true that

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

it is also true that

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

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

When comparing numbers, we must therefore be careful whether we are comparing numbers in actual numerical form or as text strings.

13.3.2

Choose the correct statement

- '16' <| '20'
- '16' > '20'
- '16' = '20'

13.3.3

Choose the correct statement

- '160' <| '20'
- '160' > '20'
- '160' = '20'

13.3.4

Choose the correct statement

- '110' > '1001'
- '110' <| '1001'
- '110' = '1001'

13.3.5

Choose the correct statement

- '333' > '3033'
- '333' <| '3033'
- '333' = '3033'

13.3.6

Choose the correct statement

- 333 <| 3033
- 333 > 3033
- 333 = 3033

13.3.7

What is the result of the following program?

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

13.3.8

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

Given that the number of digits is relatively limited when using numbers, we will use a string to load long numbers.

The procedure will be quite simple:

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

```
number = input()
max = '0'    # we will work with characters, so max will also
be saved as a character
for digit in the number:
  if max < digit:   # if the current digit is greater than the
largest so far
     max = digit    # we will remember it
print(max);
```

13.3.9

Complete the code to find the smallest digit in a number.

```
number = input()
min = '_____'   # set the largest possible value
for digit in _____:
 if min _____ digit:
    min = _____
print(min)
```

13.4 Special characters

🕮 13.4.1

In addition to regular characters, special characters are sometimes used in pritnouts. They can be used to print some characters and adjust the layout of the text.

Typical special characters are:

- \' – inserts an apostrophe into the text,
- \" – inserts quotation marks into the text,
- \\ – inserts a backslash into the text.

When using e.g.:

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

prints

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

> We used the quotation marks to delimit the string as well as inside its content. However, thanks to writing it in the form of \" there was no error.

13.4.2

Fill in special characters for text output

```
Characters ' a " are used when working with strings.

print('Characters _____ a _____ are used when working with
strings.')
```

🕮 13.4.3

The second group of special characters is used when formatting the output:

- \n - newline 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.

E.g.:

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

prints

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

The **\n** character moves the cursor ensuring text output to a new line, and the output after **\n** continues on a new line.

13.4.4

Add spaces or a newline character to get the following output:

```
Mother has a:
butter,
ice-cream
and cakes.
```

```
print('Mother _____has_____a:_____butter,_____ice-
cream_____and_____cakes.')
```

- \n
- \n
-
- \n
- \n
-
-
- \n
-
-
-
- \n

13.4.5

The \t character is used as a tab - it indents the text following it to the intended nearest tab position.

E.g.

```
print('Mother: \tteacher')
print('Father: \tclerk')
print('Daughter: \tstudent')
print('Brother: \tstudent')
```

Prints:

```
Mother:   teacher
Father:   clerk
Daughter: student
Brother:  student
```

13.4.6

```
I am
'Python'
        programmer
```

Add special symbols to the code to get the above listing.

```
print("I am_____ _____ Python_____ _____ _____ programmer")
```

- \n
- \"
- \b
- \'
- \s
- \n
- \n
- \s
- \'
- \t
- \t
- \t
- \n
- \b
- \'

13.4.7

Sometimes it is necessary to use more tabs for correct indentation.

Note that the tab replaces a maximum of 8 spaces. I.e. the character after the tab always starts at position 8*x + 1 (9,17,25, etc.).

E.g.:

```
print('my dog:\tKejsy')
print('the dog at the neighbors:\tZahraj')
```

Due to the long text at the beginning of the second line, the output has the form:

```
my dog:   Kejsy
the dog at the neighbors:   Zahraj
```

After adding the tab:

```
print('my dog:\t\tKejsy')
print('the dog at the neighbors:\tZahraj')
```

we get the desired:

```
my dog:                     Kejsy
the dog at the neighbors:   Zahraj
```

13.4.8

Provide the output:

```
first number:    1258
second number:   2257
line:            ----
total:           3515
```

```
print('first number:_____1258_____second
number:_____2257_____line:__________----
_____total:__________3515')
```

- \n
- \t
- \t
- \t
- \t
- \n
- \t
- \t
- \t
- \n
- \n
- \t
- \n

13.5 Special printouts

📖 13.5.1

In the **print()** command, in addition to the texts printing, we can also use special settings that allow changing the form of the output.

Let's mention the two most common here:

- **sep**, determines which character is inserted as a string separator in the print() command. The default setting is a space.
- **end**, determines which character is inserted at the end of the written text. By default, the character **\n** is set, which is a wrap - moving the cursor to a new line.

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

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

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

```
one
two
three
```

If we enter a semicolon as separator,

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

we get:

```
one;two;three
```

If necessary, the separator can also contain more characters, e.g. '; '

📝 13.5.2

Complete the print() command to get a printout in the form

1*2*3*4

```
print('1', '2', '3', '4', _____ = '_____)
```

📖 13.5.3

The **end** parameter in the output specifies which character or string is supposed to be printed after all the text entered in the **print()** command has been printed.

By default, it is set to \n, which causes each statement to move the cursor to a new line at the end.

By changing the **end** parameter, we can ensure that the output cursor does not move to a new line after each printout. This way, we can print texts from several commands in one line.

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

Comma + space is used as the termination string in the first two statements...

```
Hi, long time no see, how are you?
```

... and the cursor does not move to a new line.

The last printout does not have the end parameter changed, so there will be delineation, which would be reflected in the next run.

13.5.4

Complete the ending characters of the outputs so that you get the following statement:

```
I have to bring: 10 bags of flour, 3 bags of sugar, 480 eggs,
5 liters of water and a large saucepan.
```

```
print('I have to bring', _____ = '_____')
print('10 bags of flour', _____ = '_____')
print('3 bags of sugar', _____ = '_____')
print('480 eggs', _____ = '_____')
print('5 liters of water', _____ = '_____')
print('and a large saucepan', _____ = '_____')
```

- end
- end
- .
- ,
- sep
- ,
- sep
- ,
-

- ,
- sep
- sep
- end
- end
- :
- ,
- sep
- :
- ,
- .
- end
- end
- sep
- ,

13.5.5

What is printed on the output? Pay attention to each character of the printout.

```
print("1","2","3","4", sep = "*", end = "!")
```

13.5.6

For some printout, the division of the text into immutable (static) strings and variable values (variables) can complicate the clarity of the notation.

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

This seemingly confusing output returns:

```
The sum of 10 and 20 is 30 .
```

There is also a space before the period at the end of the sentence, which is not in accordance with the rules of writing the text

For such structured statements, Python provides a simpler form of notation.

```
print(f'The sum of {x} and {y} is {z}.')
```

This notation before the text in quotes or apostrophes itself contains the letter f, which lets the compiler knows that the content of the following string should be modified so that instead of the content of the brackets {}, it inserts the value of the relevant variables.

The result of the output therefore will be:

```
The sum of 10 and 20 is 30.
```

Every space and every character entered inside the format string will also be reflected in the output.

13.5.7

Fill in the correct code so that we receive the exact required output.

```
j = 25
k = 12
print(_____'_____j_____ - _____k_____ = _____j-k_____')
```

The required output is

```
25 - 12 = 13
```

13.5.8

Print the multiplier for the number entered in the input in the form (e.g. for 5):

```
1*5=5
2*5=10
3*5=15...
```

Complete the program that ensures the printout in the required form:

```
text = input('Enter a value: ')
n = _____(text)
for i in range(1,11):
    print(_____'_____*_____=_____i*n_____')
```

13.6 Working with characters (programs)

13.6.1 Word order

Write a program that, for three strings entered on separate lines, finds their alphabetical order and lists them alphabetically below each other.

```
Input : Adam
Jano
Eva
Output:
Adam
```

```
Eva
Jano
```

```
Input : beta
Alfa
Simon
Output:
Alfa
Simon
beta
```

⌨ 13.6.2 The number of lowercase letters

Write a program that finds the number of lowercase letters in a sentence without diacritics given as input.

```
Input: Mama has Ema.
Output: 7
```

⌨ 13.6.3 Printing part of the ASCII table

Write a program that, for two given numeric values, prints the ASCII table characters located at the positions between them. If the values are outside the range of 33 and 127, it will print "error". Assume that the first value entered is less than the second.

Write characters in the form of position, tab, character.

```
Input :
51
56
Output:
51    3
52    4
53    5
54    6
55    7
56    8
```

```
Input : 21
120
Output: error
```

⌨ 13.6.4 ASCII encoding

Encode the entered text by using their ASCII values instead of characters and separating them with commas in the output.

```
Input: Adam
Output: 65,100,97,109
```

⌨ 13.6.5 ASCII decoding

Write a program that decodes a given word using its ASCII codes.

The input starts with the number of characters to be decoded, followed by an integer value representing the character code in each line.

Print the decoded string in a line.

```
Input:
4
65
108
101
120
Output: Alex
```

⌨ 13.6.6 Dictation control

For the typed text submitted by the student, check the number of errors by comparing it with the teacher's sample word. For the input representing the student's text in the first line and the teacher's text in the second, write the number of times the student wrote the wrong character. Before checking the text itself, check whether the submitted texts have the same number of characters, and if not, end the solution with the message: "different number of characters".

If the texts are identical, write: "no errors", otherwise "the number of mistakes: x".

```
input: Word
word
output:
the number of mistakes: 1
```

```
input: word
word.
output:
different number of characters
```

⌨ 13.6.7 Encoding II.

Encode the text by shifting the individual characters two positions to the right in the ASCII table.

```
Input: mama
Output: ococ
```

⌨ 13.6.8 Deleting the numbers

Write a program that replaces digits with dashes in the given string. The letters will remain unchanged.

```
Input : Hello123
Output: Hello---
```

```
Input : 123
Output: ---
```

```
Input : hello 0john
Output: hello -john
```

⌨ 13.6.9 Sum of numbers

Write a program that calculates the sum of the integers appearing in a string. A decimal number is taken as 2 separate numbers.

```
Input : We have 12 hens, 54 gooses and 3 dugs.
Output: 69
```

```
Input : 12.3,8 9
Output: 32
```

Slices and Basic Functions

Chapter 14

14.1 Slices

📖 14.1.1

By using the square brackets, we can get to any character in the string. For example for the string

```
ret = 'Sagarmatha'
```

this entry

```
x = ret[2]
```

will store character **g** to the variable **x**.

In addition to getting one character, we can also read several characters from the string at once. The entry we refer to as **slice** is used for this. A slice is created by specifying a variable name and a definition for a character selection of the form:

```
x = ret[beginning : end]
```

The slice limits are determined similarly to generating a list via **range()**. Unless we enter otherwise, step is set to 1.

E.g.

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

prints the text **ga** based on the fact that it starts at position 2 and ends one position earlier than the specified value for end (so it takes the 3rd character as the last). Therefore it reads the characters **g**, **a**.

📝 14.1.2

What is the result of the following code?

```
ret = 'Montevideo'
x = ret[3:6]
print(x)
```

📝 14.1.3

Complete the results of the output:

```
ret = 'Marvel Universe'
```

```
print(ret[0:len(ret)])   # prints _____
print(ret[0:6])          # prints _____
print(ret[7:len(ret)])   # prints _____
```

14.1.4

Complete the correct slice boundaries for the **Com** output.

```
ret = 'DC Comics'
print(ret[_____:_____])
```

14.1.5

Similar to **range()**, we can also use a step when cutting. The entry then has the form

```
x = ret[Beginning : end : step]
```

E.g. for

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

258 is printed.

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

14.1.6

What does the following code print?

```
ret = '0123456789'
x = ret[2:9:2]
print(x)
```

14.1.7

What does the following code print?

```
ret = 'Good afternoon!'
x = ret[2:12:4]
print(x)
```

📖 14.1.8

If we omit a parameter in the notation of the slice, Python will automatically fill it in.

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

The version with step indication also works.

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

it copies every second character of the string.

📝 14.1.9

Fill in the slice parameters so that the **x** variable contains the texts mentioned in the comments:

```
ret = 'Good afternoon!'
x = ret[_____ : _____ : _____]  # noon

ret = '0123456789'
x = ret[_____ : 8 : _____] # 036
```

📝 14.1.10

What does the following code print?

```
ret = '0123456789'
x = ret[5:]
print(x)
```

📝 14.1.11

What is stored in the variable x after the following code is executed?

```
ret = '0123456789'
x = ret[:2]   # variable x contains _____
x = ret[::3]  # variable x contains _____
```

14.2 Negative indexes

📖 14.2.1

In addition to classic indexing, Python also allows indexing with negative values.

Negative indices start numbering from the last character that has an index of -1. We proceed from the last character to the first, so that the second character from the end has an index of -2, and so on.

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

📝 14.2.2

What character is inserted into the variable x in the following steps of the program?

```
ret = 'Priscilla'
x = ret[-1]  # x contains _____
x = ret[-5]  # x contains _____
```

📖 14.2.3

Negative values can be also used as part of a slice. They can define the beginning of the sequence and the end of the sequence. The principle of evaluating such a slice consists in replacing a negative value with a real index.

If the end value is less than the start value, the result is empty.

E.g.

```
ret = '0123456789'
x = ret[-1 : -3] # it means that it will start on the last
character, i.e. index 9 and ends at the third from the end of
t. j. 7
print(x)
```

prints an empty string.

However, if we enter:

```
ret = '0123456789'
```

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

the characters from the 3rd from the end (**7**) to the 1st from the end (**9**) will be printed, which will no longer be included in the printout, i.e.: **78**

14.2.4

What will the variable x contain after each step?

```
ret = 'Altavista'
x = ret[-6:-2]  # x contains _____
x = ret[-7:]    # x contains _____
x = ret[:-5]    # x contains _____
```

14.2.5

A negative value in a step has a special meaning. In such a case, during the selection of characters, the procedure is by going from a larger index to a smaller one. Thus, the index in the first position must be greater than in the second.

E.g.

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

prints **omol**.

14.2.6

What will the variable x contain after each step is executed?

```
ret = 'Gargantua'
x = ret[4:0:-1] # x contains _____
x = ret[6:2:-1] # x contains _____
x = ret[7:2:-2] # x contains _____
```

14.2.7

We can also replace indexes with their negative values

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

The output will be **omol**.

14.2.8

What will the variable x contain after each step is executed?

```
ret = 'Pantagruel'
x = ret[-2:-4:-1]   # x contains _____
x = ret[-5:-8:-1]   # x contains _____
x = ret[-1:3:-2]    # x contains _____
```

14.2.9

If we omit the slice values in the notation and set only a step, Python will complete them by starting with the last and ending with the first character in the case of a negative step. It prints the text in reverse order - the result starts from the last character.

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

prints **nomoloS**.

It always starts generating the result from the last character.

14.2.10

What will the variable x contain after each step is executed?

```
ret = 'Halikarnas'
x = ret[::-1]   # x contains _____
x = ret[::-2]   # x contains _____
x = ret[::-3]   # x contains _____
```

14.3 Basic functions

14.3.1

Let's imagine a test in which we need to verify the correctness of the answer to a question

How many countries in the world have more than 500,000 inhabitants

Although the answer looks obvious at first glance, the student can answer in several ways:

```
2, 02, two, Two, TWO
```

For short answers, we can enter a condition with all the possibilities that can occur, but if we think about their complexity, it is better to take a different approach. We would have to treat:

- all combinations of upper and lower case letters (two, Two, Two, TWO, tWO, twO, TwO, tWo),
- checking for all numbers of zeros before numbers (2,02,002, etc.)
- checking for unnecessarily entered spaces before or after a word (' 2','2 ')

14.3.2

How many different combinations of uppercase and lowercase letters can be made in the word **three**?

14.3.3

If we think about the step-by-step processing of the answer, the step-by-step processing could consist of the following steps:

- remove spaces before and after the text from the input,
- if the input consists of only digits, convert it to a number and see if it's correct - leading zeros are ignored in that case,
- convert all letters to the same form - either all lowercase or all uppercase and compare with the correct answer.

14.3.4

Complete the command that will convert the text to a number:

```
text1 = input()
a = _____(text1)
```

14.3.5

The **strip() function** provide the removal of spaces from the end and beginning of the string. This **function** is used as part of a string variable - we separate it from the variable name with a period.

```
ret = ' Mama has Ema at home.  '
cleaned = ret.strip() #  obsahuje 'Mama has Ema at home.'
```

The function does not remove spaces from inside the text, only from the edges.

14.3.6

What will be stored in the **answer** variable after the **strip()** command is executed?

```
ret = ' a b c 123 . '
answer = ret.strip()
```

14.3.7

In the next step, we should check if the cleaned string consist only of digits.

To determine the type of characters that a string contains, we use the following string functions:

- **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 a string are letters; if so, it returns **True**, otherwise it returns **False**.
- **ret.isalnum()** tests whether all characters in a string are letters or numbers; if so, it returns **True**, otherwise it returns **False**. What other characters can be in the string? For example space, comma, parentheses, etc.

E.g. for

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

False will be printed because the string contains a dot character in addition to numbers.

For

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

True will be printed because the string contains only numbers and letters.

> Attention, a space is considered a special character - neither a number nor a letter.

14.3.8

Fill in the correct results for the use of the functions

```
ret = '012540'
print(ret.isdigit())   # prints _____
ret = 'abCD'
```

```
print(ret.isalpha()) # prints _____
print('10.59'.isdigit()) # prints _____
print('a10'.isalpha()) # prints _____
print('a10'.isdigit()) # prints _____
print('a10'.isalnum()) # prints _____
print('Pozor!'.isalpha()) # prints _____
print('a b c'.isalnum()) # prints _____
print('3 children'.isalnum()) # prints _____
```

- False
- True
- False
- False
- True
- True
- True
- False
- False
- False
- False
- True
- False
- False
- True
- False
- True

📖 14.3.9

In addition to the type, we can also check the case of the letters in the string:

- **ret.islower()** tests whether all characters in the string are from the set of lowercase letters,

ret.isupper() tests whether all characters in the string are from the uppercase set.

When executing the function, only letters are checked . Other characters are not considered, therefore e.g.

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

prints **True**.

📝 14.3.10

Fill in the correct results for the use of the functions

```
print('Alpha'.islower())  # prints _____
print('BETA'.isupper())   # prints _____
print('BETA'.islower())   # prints _____
print('Var'.isupper())    # prints _____
print('Attention!'.isupper()) # prints _____
```

- False
- False
- True
- True
- False
- True
- False
- True
- False
- True

📖 14.3.11

In programs we usually don't waste time by checking whether a string contains all lowercase or uppercase letters, but we just convert it to lowercase or uppercase. The **lower()** and **upper()** functions are used for this.

ret.lower()returns all letters changed to lowercase as a result. The original string remains unchanged, e.g.:

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

ret.upper() returns all uppercase letters as a result. The original string remains unchanged, e.g.:

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

📝 14.3.12

Fill in the correct results for the use of the functions

```
print('Alpha'.lower())  # prints _____
print('BETA'.upper())   # prints _____
print('BETA'.lower())   # prints _____
print('Var'.upper())    # prints _____
```

```
print('POZOR!'.lower()) # prints _____
```

14.3.13

Since we already know all the necessary functions to handle the task from the beginning of the lesson, add the correct commands to verify the correctness of the answer:

```
print('How many countries in the world have more than 500,000
inhabitants?')
text_o1 = input()
# remove spaces before and after the text from the input
text_o2 = text_o1._____()
# if the input consists only of digits,
if text_o2._____():
   # I will convert it to a number
   number_o3 = _____(text_o2)
   # and check if it's correct - leading zeros are ignored in
that case
   if cislo_o3 == 2:
       print('correct_____)
   else:
       print('incorrect')
else:
   # the input does not contain only digits
   # convert the string e.g. to lowercase (I could also use
uppercase)
   text_o4 = text_o2._____()
   # compare with the correct answer consisting of lowercase
letters
   if text_o4 == '_____':
       print('correct')
   else:
       print('incorrect')
```

14.4 Functions in string (programs)

14.4.1 Comparing the number of digits

Write a program that, for two given strings, prints the number of digits in them and decides which contains more digits. Use the functions from this chapter.

```
Input: we have 72 hens
we have 3 rabbits
```

```
Output: 2 1 string1
```

```
Input: we have 72 hens
we have 3 rabbits and 2 pigs
Output: 2 2 match
```

⌨ 14.4.2 Upper and lowercase letters

Write a program that checks whether two entered strings are the same - it will not take their case into consideration, so mama and MaMa represent the same string. Also make sure to ignore spaces before and after the entered text.

```
Input: Mother
MOTHER
Output: match
```

```
Input: Father
Dad
Output: mismatch
```

⌨ 14.4.3 Number of letters in the string

Write a program that, for a entered string, finds the number of individual letters (a-z) in it. The user is required to use lowercase letters. If he also enters capital letters or numbers, write "error".

The number of letters used is displayed only if the given letter occurs in the string. The list of characters must be in alphabetical order.

```
Input: mama
Output:
a-2
m-2
```

```
Input: winter
Output:
e-1
i-1
n-1
r-1
t-1
w-1
```

While Loop

Chapter 15

15.1 While

📖 15.1.1

Sometimes when using a loop, we don't know how many times it will need to be repeated. However, we can determine the condition until when the loop should be repeated. For example: while you are hungry, eat a cookie.

In such a case, we can ensure the execution of the loop through the **while** command and the condition, the fulfillment of which will ensure the execution of the commands in the body of the loop. Its structure is similar to the **if** statement. The difference is that the commands contained in the **while** loop are repeated **until** the condition is met (evaluated as **True**).

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

📝 15.1.2

What keyword (statement) defines a loop with a condition at the beginning?

- while
- for
- if

📖 15.1.3

The loop works like this:

1. verifies the validity of the condition
2. if the condition is met, the block of commands is executed,
3. execution will return to point 1.

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

The output has the form:

```
1
2
3
4
5
```

Note that in the block of commands, it is necessary to change the variable that is tested in the condition. If we didn't do that, the condition would be fulfilled all the time and the loop would go on endlessly. In our case, we had to increase the value of x by 1.

15.1.4

Write "Hello" 10 times below each other

The task is practically the same as in the case of using the for loop. We can write any task that requires repetition of commands through any type of loop, and it is up to us which type of loop we choose.

In this case, the programmer must provide all the operations contained in the **for** structure in separate commands:

setting the initial value of the control variable,

- the condition that determines the end of the loop,
- execution of commands in a loop,
- increasing the value of the control variable.

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

- 1
- 0
- i-1
- 11
- 10
- i + 1
- -1

15.1.5

Arrange the code so that the loop outputs the numbers from 0 to n.

```
n = int(input('Enter n:'))
```

- counter += 1
- print(counter)
- while counter <= n:
- counter = 0

15.1.6

Complete the code so that 5 dots are printed in one line in a row:

```
i = 4
_____ i <= _____:
  print('.', _____ = '')
  i = i + 1
```

15.1.7

Using the while loop, write even numbers from the interval from 8 to 24 below each other.

We will print out the contents of the variable whose value will be increased by 2 in each step of the cycle.

We will execute the activity until the value reaches 24.

```
number = 8
while number <= 24:
   print(number)
   number = number + 2
```

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

```
for number in range(8,25,2):
    print(number)
```

15.1.8

Complete the program so that it prints all numbers divisible by ten that are less than the number entered in the input.

```
max = int(input('Enter the upper limit: '))
number = 0
_____ number < _____:
  print(_____)
  number = number + _____
```

15.1.9

The **while** loop is referred to as a safe loop, because it is tested before the action is executed. If the condition is not met at the first verification, the loop commands may run not even once. For example:

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')
```

In this case, no number is printed, because the condition is not met during the first verification - **i** (10) is not less than the **upper** (also 10).

For other values of the upper and lower limits, of course, the loop can be executed.

15.1.10

For which pairs of values nothing is printed(that is, the loop does not run)?

```
a = int(input())
b = int(input())
i = a
while i >= b :
   print(i)
   i = i - 1
```

- 5, 8
- 8, 5
- 5, 5
- 8, 8

15.2 Break command

15.2.1

Complete the program that finds the number of divisors of the entered number and prints them.

The divisor is the number by which, when we divide the tested number, we get a remainder of zero. So it makes sence to examine the numbers from 1 to the entered number.

Let's do a research using a **while** loop:

```
a = _____(input('Enter a number: '))
i = _____
number = _____
_____ i _____ a :
  if a _____ i _____ 0:
      print(_____)
      number = _____
  i = i _____ 1
print('The number of divisors is', _____)
```

- 1
- >=
- +
- 1
- 0
- >
- int
- number - 1
- <|
- -
- *
- while
- i
- number * 2
- ==
- %
- ==
- number + 1
- number
- <=

📖 15.2.2

Sometimes the algorithm is written in such a way that it is convenient to end the loop earlier than it would end in the "natural" way. The **break** command is used to interrupt the loop and continue the execution of commands after 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 **i** values and if it exceeds 20, it ends the loop with the **break** command - 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')
```

15.2.3

What statement terminates a loop regardless of what stage of execution it is currently in and ensures that the program continues after the loop?

15.2.4

Complete the program that determines whether the entered number is a prime number.

It is true that a number is prime if it has only two divisors, namely 1 and the number itself. We could solve the problem by counting all its divisors as in the previous problem.

However, we know that a number is not prime as soon as we find the first divisor other than 1 or the number itself. Then there is no point in continuing the research, because it is useless.

However, we must remember the information that we have found a divisor so that we can write a message to the user at the end of the program based on this information. If a divisor is found, we store the value 1 in the variable **count** and end the execution of the loop. This is done by the **break** command, which definitively ends the loop and the program continues with the commands after the loop.

In case the loop that starts with the value 2 and ends by examining a value 1 less than the entered number reaches the end without finding another divisor, the value 0 is stored in the variable **count**.

```
n = int(input('Enter a number: '))
number = 0
i = 2                    # the examination will start from
value 2
```

```
while i _____ n:          # it will run until i < n
   if n _____ i == 0:    # if n is divisible without a
remainder, we have a divisor
      number = 1          # set the count to 1
      _____              # end the execution of the loop
   i = i + 1

# here it is continued after the end of the loop or after the
command _____
if number _____ _____:
   print('it is prime number')
else:
   print('it is not prime number')
```

- <|=
- stop
- +
- %
- exit
- <
- exit
- 0
- break
- continue
- continue
- break
- 1
- ==
- stop

📖 15.2.5

However, the program from the previous task could be also written more simply. We could write the divisibility condition directly into the loop condition - the cycle would ensure the increase of the examined value until the remainder after division was zero. This situation will certainly occur and at the latest it will occur if **i** has the value **n**.

Based on the value of i, we would decide whether the divisor was found before it reached the value of **n**.

```
n = int(input('Enter a number: '))
i = 2                  # the examination starts from value 2
while n % i != 0:      # until a divisor is found
   i = i + 1           # moving on to explore the next issue
```

```
if i == n:          # if I got to the number itself, it is a
prime number
   print('it is a prime number')
else:
   print('it is not a prime number')
```

15.2.6

Fill in the program that detects whether there is a number divisible by 17 in the entered interval.

```
lower = int(input('enter the lower limit: '))
upper = int(input('enter the upper limit: '))
i = _____
while _____ <= _____:
   if i _____ 17 == 0:
      _____
   i = i + 1

if i > upper:
   print('_____')
else:
   print('_____')
```

- if
- break
- lower
- %
- upper
- lower
- there is
- exit
- upper
- while
- there is not
- i
- /

15.2.7

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

We can verify the existence of the character in the condition of the loop, and then, when the character **'s'** is found, write that we have found it and end the loop.

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

Which of the following statements are true?

- There are no mistakes in the program.
- The program crashes for strings that do not contain the 's' character.
- The program is infinite for some strings.
- The program works flawlessly for words starting with 's'.
- The program works flawlessly for words ending with 's'.

15.2.8

The problem in the previous program occurs if we try to read a character at a position beyond the end of the string. **There is no** such character and the program would terminate with an error.

The solution is to evaluate two conditions: we add a condition to check whether we are not already past the last character:

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

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

If the loop has been completed, we will find out in what way:

- if **i** is less than the number of characters in the string, it means that the cycle ended before the condition that we are past the last character of the string was true - that is, the character 's' was found
- otherwise, the loop ended if the condition that i < len(ret) was not fulfilled - that is, the end of the string was reached and nothing was found.

- >
- <|=
- <
- >=

- or
- !=
- not
- len
- if
- and
- ==

15.2.9

When creating a condition in the loop, you should consider that if **i** has reached the value **len(ret)**, then an attempt to read the character **ret[i]** will end with an error.

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

Therefore, we write the condition by first checking if i < len(ret).

- If yes, the evaluation of the condition continues.
- If not, the evaluation of the condition will **end**, because the result (of two conditions that should apply simultaneously) will be **False**, regardless of the result of the second part of the condition - so there will be no reading of a character outside the string.

If parts of the condition were reversed, the program would **crash** whenever it went past the last character of the string and tried to compare it to 's'.

15.2.10

Write a program that detects whether a number entered as a string contains the digits 4 or 8. If so, print which one was found first.

Arrange the conditions correctly and complete the code.

```
ret = input()
i = 0
while (_____) and (ret[i] != '4') and (_____):
  i = i + 1
if i _____ _____:
   print('was not found')
else:
   print('as first was found the digit',_____)
```

- <|=
- i < len(ret)

- ret[i]
- <|
- i
- >
- ==
- ret[i] != '8'
- len(ret)
- ret[i-1]

15.2.11

And let's try to solve the same problem in another way:

Write a program that determines whether a number entered as a string contains the digits 4 or 8.

Complete the code:

```
ret = input()
i = 0
while i < len(ret):
  if (ret[i] _____ '4') _____ (ret[i] _____ '8'):
     print('was found', ret[i])
     _____
  i = i _____ 1
if i _____ _____(ret):
   print('was not found')
```

15.2.12

Previous programs found only the first occurrence of the searched value. Now let's try to find all occurrences.

Write a program that detects whether a number entered as a string contains the digit 5 and prints the positions of all its occurrences. If it does not find any occurrence, it informs the user about it.

In order to have information about whether the value 5 occurred at least once after the end of the loop, we use a **boolean** variable. We set it to **False** at the beginning and change it to **True** when 5 occurs. After the end of the loop, we will be able to identify whether it is necessary to print that it was not found or do nothing (because the positions of the occurrence of the number 5 have been printed).

Complete the code:

```
ret = input()
i = 0
```

```
found = _____
while i < len(ret):
  if ret[i] _____ '5':
     print('position', i)
     found = _____
  i = i + 1
if _____ found:
   print('was not found')
```

- true
- False
- ==
- True
- !=
- True
- false
- not
- is
- False
- or

15.3 Infinite loop

🕮 15.3.1

Although we have mentioned several times that an infinite loop is undesirable for program execution, some tasks are easier to write using it and, in a special case, jump out of it using the **break** command.

The easiest way to write an always true condition is with **True** (we don't use the "calculation" of the condition, but just write the result):

```
while True:
    command
```

And the moment we achieve the desired results, we can end the loop.

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

15.3.2

Complete the condition so that the loop is infinite:

```
while _____:
    command
```

15.3.3

An infinite loop is often used when testing the value of an input variable.

If we want the user to enter a positive integer, we should also check it. In case he did not fulfill the request, we ask him again and again and again until the program gets the required value. If the user keeps entering a negative value, we can continue ad infinitum.

After entering the correct value, we interrupt the cycle and the program continues after the loop.

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

15.3.4

Complete the program so that it receives a negative value from the user.

```
while _____:
  n = int (input('Enter negative number: '))
  if n _____ 0:
    _____
print('Thanks.')
```

15.3.5

How many times is the word Python printed?

```
i = 1
while True:
    print('Python')
    i += 1
    if (i > 10):
        break
```

📖 15.3.6

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

We can modify the program for obtaining a positive number as follows:

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

We used input loading in two places.

The cycle is activated only if we first entered an incorrect value.

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

📝 15.3.7

Complete the program so that it force the user to enter an even number:

```
n = int(input('Enter an even number: '))
_____ n _____ 2 == _____:
   n = _____(input('Enter an even number again: '))
print('We can continue')
```

📝 15.3.8

Write a program to find the average marks in an arbitrary subject. We will not enter the number of marks at the beginning, but we will end the program by entering the value 0.

Since we don't know the number of grades, we need to use a loop that will keep being executed until a value of 0 is entered. This value is no longer included in the average.

In order to be able to calculate the average, we also need to know the number of grades - we will use a counter.

```
print('Give me grades.')
sum = _____              # set the starting value for the sum
number = _____           # set the initial value for the number
while _____:         # use an infinite loop
 _____ = int(input('Enter the mark: '))
 if mark == _____:    # if termination with 0 was entered
```

```
    _____             # stop loading
 sum = sum + _____    # add the mark
 number = number + _____        # increase the number
print('The average mark is ',sum_____ number)
```

- 0
- 0
- mark
- 0
- /
- 1
- break
- number
- 1
- 1
- digit
- True
- mark
- 0
- sum

15.4 While (programs)

⌨ 15.4.1 Capital letter search

Write a program that detects as quickly as possible whether there is an uppercase letter in the given string and prints the position of its first occurrence. If it does not find any occurrence, it informs the user about it.

```
input:
attention OSBD
output:
exists: 6
```

```
output:
attention winter is coming
input:
does not exist
```

⌨ 15.4.2 Input control

Write a program that finds the greatest common divisor of two positive numbers. Make sure the program keeps asking for the value again if the user enters a non-positive number.

```
input:
10
20
output:
10
```

```
input:
-10
-5
6
-6
8
output:
2
```

⌨ 15.4.3 Highest salary

Write a program that finds the highest value in the salary list of school employees. The number of employees willing to disclose their salary is not known at the beginning. We end the loading by entering the value 0.

```
input:
1100
950
980
1121
830
0
output:
1121
```

⌨ 15.4.4 Tired tourist

Write a program that, based on the ascents and descents in meters of altitude expressed as positive and negative values, determines whether the tourist's destination point is higher or lower than the starting point and by how much. The number of ascents and descents is unknown in advance, the list is terminated by zero. If there is no height difference between the starting point and the destination point, a match is displayed.

```
input:
100
-50
30
-10
-10
```

```
0
output:
higher by 60
```

```
input:
-50
30
-10
-10
0
output:
lower by 40
```

Simple Lists

Chapter 16

16.1 Several variables

📖 16.1.1

Python has its own specifics for working with variables. One of them is the ability to assign values to multiple variables in one command.

```
a, b, c = 10, 20, 30
```

This entry assigns to the first variable (**a**) the first value given after the "=" sign, i.e. **10**, to the second variable (**b**) the value of **20**, etc.

📝 16.1.2

What is printed after this sequence of commands is executed?

```
x, y = 20, 30
x, y = y, x
print(x, y)
```

- 30 20
- 20 30
- 20 20
- 30 30
- error

📖 16.1.3

It is also possible to set multiple variables to the same value in one line.

```
x = y = z = 0
```

The assignment goes from right to left, first **0** is assigned to the variable **z**, then **y** is assigned the value **z**, and at the end **z** is assigned the value **y**.

📝 16.1.4

What is printed after this sequence of commands is executed?

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

- 30 10 10

- 20 30 10
- 30 20 10
- 10 20 30
- 10 10 30
- 20 10 30
- 10 30 10
- 30 20 20

16.1.5

When retrieving input from the user, there are frequent situations where we need to retrieve more than one value. Loading e.g. of three values (names) could look as follows:

```
ret1 = input('Enter the 1st name: ')
ret2 = input('Enter the 2nd name: ')
ret3 = input('Enter the 3rd name: ')
```

However, there is also an option to load all three names at once - by entering them in one line.

Strings have a **split()** function that can split text into multiple parts based on the use of a space. Text

```
Ivan Michal Zuzana
```

can be split into 3 different values that are put into three different variables

```
ret1, ret2, ret3 = 'Ivan Michal Zuzana'.split()
```

The entire load would then look like this:

```
ret = input('Enter three names separated by a space: ')
ret1, ret2, ret3 = ret.split()
```

> Attention, if the string is not divided into the correct number of words, the program will crash.

16.1.6

Complete the program code that reads and prints the four input words:

```
_____ = input('Enter 4 words separated by a space: ')
_____,_____,_____,_____ = ret._____()
print('1st value:',u)
print('2nd value:',o)
print('3rd value:',h)
print('4th value:',m)
```

- h
- m
- u
- split
- o
- o
- ret
- u
- ret
- h
- split()
- m

16.1.7

If our list of first names contained persons with multi-word first names (e.g. Milan Rastislav, Adam Ivan, etc.), using a simple split() would be problematic due to the large number of spaces.

For such purposes, the **split()** function also has a form in which we can enter the dividing character. It can be used as follows:

```
ret = input('Enter 4 names separated by a comma: ')
a,b,c,d = ret.split(',')
```

We then enter the input in the form where individual names or double names are separated by commas. Attention, the space after the comma is counted as a normal character in this case.

```
Adam,Beata Anna,Jozef Francis,Ivan
```

16.1.8

Complete the code of the program that determines which of the four entered words separated by commas at the input has the most characters (assume that they have a different number of characters).

```
ret = input('Enter 4 words separated by a comma: ')
a, b, c, d = ret.split_____
max = _____
if len(b) > len(max):  # if the 2nd word has more characters
than the current max
    max = _____              # it becomes the new max
if len(c) > len(max):  # if the 3rd word has more characters
than the current max
    max = _____              # it becomes the new max
```

```
if len(d) > len(max):  # if the 4th word has more characters
than the current max
    max = _____             # it becomes the new max
print(_____)
```

- "
- ()
- b
- split()
- c
- max
- (',')
- c
- ret
- a
- d
- a
- a
- b
- d

16.2 List in a loop

📖 16.2.1

We can also loop through string values. This way we can create a list that we can go through.

```
for kind in 'cat', 'dog', 'fish', 'hamster':
   print('My favourite animal is', kind)

My favourite animal is cat
My favourite animal is dog
My favourite animal is fish
My favourite animal is hamster
```

📝 16.2.2

Complete the program so that it prints favorite subjects in the following order:

```
informatics
mathematics
physics
chemistry
```

```
for _____ _____ '_____', '_____', '_____', '_____':
    print('My favourite subject is', name)
```

- physics
- split
- string
- name
- mathematics
- i
- in
- for
- chemistry
- split(',')
- informatics

📖 16.2.3

Of course, we don't have to print all the values, but we usually process them in the body of the loop.

```
for animal in 'cat', 'dog', 'fish', 'hamster':
  if animal == 'dog':
    print('My favourite animal barks')
  if animal == 'fish':
    print('My favourite animal is silent')
```

In this case, only the animal from the list that meets our processing requirements will be printed.

📝 16.2.4

What does the following program print?

```
for animal in 'cat', 'dog', 'fish', 'hamster':
  if len(animal) > 5:
     print(animal)
```

📖 16.2.5

We will now use the loop's ability to iterate over a list of words and the split() function's ability to split the input into multiple (list) variables.

If we input a list separated by spaces or commas, we can process each value separately. The following program reads a line of comma-separated values and prints each one.

```
list = input('Enter words separated by a comma: ')
```

```
for i in list.split(','):
   print(i)
```

16.2.6

Complete a program that reads a list of words separated by spaces and for each one of them prints the number of characters they contain:

```
list = input('Enter words separated by a space: ')
for _____ in list._____:
   print(word,'-',_____(word))
```

16.3 Random numbers

16.3.1

A random number is a useful tool for testing programs or introducing an element of randomness to a game or programs.

We first need to connect the random number generator to the program - import it. We do so by entering:

```
import random
```

which gives us a library (module) allowing to obtain random values.

The first basic function in the module is

```
random.randrange(end)
```

which returns a random integer from the list generated by **range()**, i.e. in the range **0** to **end -1**.

E.g.

```
a = random.randrange(20)
```

inserts one random value from the interval 0-19 into the variable **a**.

16.3.2

Complete the code so that the program generates a random number from the interval <0,15>

```
_____ random
x = _____.randrange(_____)
```

```
print(x)
```

16.3.3

The random module also includes other functions:

```
random.randrange(start, end)
```

Generates a random value from the interval start .. end-1, e.g.

```
random.randrange(-10, 11)
```

Generates a random value from -10 to 10.

```
random.randrange(start, end, step)
```

Selects a random value from the list generated by the range command(start, end, step), e.g.

```
random.randrange(-10, 11, 2)
```

Generates an even random number in the range -10 to 10.

16.3.4

Complete the code so that the program generates a random number from the interval <10,20>

```
_____ random
x = _____.randrange(_____,_____)
print(x)
```

16.3.5

Complete the code so that the program generates a random number from the interval <-25,25>

```
import _____
x = random._____(_____, _____)
print(x)
```

16.3.6

Complete the code so that the program generates a random number from the interval <-30,30> that is divisible by 5.

```
import _____
```

```
x = random._____(_____, _____, _____)
print(x)
```

- randrange
- 5
- -29
- 30
- 31
- range
- -30
- 10
- -31
- 4
- 29
- random
- rand

16.3.7

In case we want to generate decimal (real) values, we need to use:

```
random.random()
```

which generates a random value from the interval <0.1).

The second option is

```
random.uniform(star, end)
```

which generates a random value from the interval <start, end), e.g.

```
random.uniform(-5.5, 10)
```

will generate a random number from -5.5 to 10, but will never generate the value 10.

16.3.8

Which values can be printed by the following program?

```
import random
a = random.uniform(-5, 7)
print(a)
```

- -5
- 0
- -2.57
- 4.18

- -5.2
- 7
- 7.1

📖 16.3.9

Write a program that generates a random number from 0 to 100 and allows the user to guess it. After each attempt, it guides him whether the guessed value is greater or less than his attempt.

```
import random
searched = random.randrange(0, 101)
print('I think a number from 0 .. 100')
```

In an infinite loop, we can ask for a guess.

```
attempt = 0
while True:
    guess = int(input('Your guess: '))
    attempt += 1
```

After entering the guess, we check whether it matches the number you are looking for. If so, we print information about a successful guess and end the cycle.

```
    if guess == searched:
        print(f'You got it right at the {pokus}. attempt.')
        break
```

otherwise, we tell the user whether their guess was too high or too low and return to the beginning of the loop.

```
    else:
        if guess < searched:
            print('Try larger.')
        else:
            print('Try less.')
```

📝 16.3.10

Arrange the code so that the program randomly generates a number from the range -10 to 10.

After generating the number, let the program inform whether the selected number was positive or negative.

When zero is generated, the program ends and reports the number of values generated.

```
    print(f'{c} is positive.')
  if c > 0:
    print('The zero was selected, we are done.')
attempt = 0
  elif c < 0:
while True:
    break
  c = random.randint(-10, 10)
  else:
    print(f'{c} is negative.')
    print(f'It took {attempt} generations.')
  attempt += 1
import random
```

16.4 Lists and random numbers (programs)

⌨ 16.4.1 List in line

The input contains 5 integers separated by a comma (-1000 to 1000). Write a program that prints the smallest one of them.

```
Input: -10,20,-75,16,8
Output: -75
```

⌨ 16.4.2 Searching for a name

Find how many times the name entered in the input occurs in a comma-separated list of names.

```
Input: Anna,Beta,Anna,Ivan,Jan,Samuel,Peter,Anna,Jan
Anna
Output: 3
```

⌨ 16.4.3 Highlighting

Write a program that adjusts the entered text so that all words starting with a capital letter are printed in upper case letters.

```
Input: dear Andrew, I am writing from Prague.
Output: dear ANDREW, I am writing from PRAGUE.
```

⌨ 16.4.4 Word mirror

Write a program that, for the entered text, modifies all the words by rewriting them backwards. Attention, not the whole sentence, each word separately.

```
Input: Dear Andrew, I am writing from Prague.
Output: raeD ,werdnA I ma gnitirw morf .eugarP
```

⌨ 16.4.5 A random number from 0 to 100 interval

Write a program that generates and prints to the console a random integer from the interval <0.100>. For example:

```
Output: 42
```

⌨ 16.4.6 A random number from the -50 to 50 interval

Generate and print a random integer <-50,50>:

```
Output (e.g.): -5
```

⌨ 16.4.7 A random number from the entered interval

Generate and print a random integer for the interval specified by a pair of integer values on the input separated by a space. Values do not have to be entered in the order from smaller to larger.

```
Input: 20 80
Output(e.g.): 61
```

```
Input: 22 -68
Output(e.g.): -3
```

Working with Strings

Chapter 17

17.1 Nested loop

📖 17.1.1

We can solve many problems using a single loop, but it is not unusual if the solution requires us to use a loop in the body of another loop. We call the inside loop a **nested loop**.

It has the form:

```
for i in range(10):
   for j in range(5):
      command
```

However, when combining multiple loops with a known number of repetitions, care must be taken to ensure that the control variables have **different names**.

📝 17.1.2

How do you refer to a loop inside another loop?

- nested
- intern
- integrated
- hybrid

📖 17.1.3

Write a program that prints one character 1 in the first line, two characters 2 in the second, and so on until 9.

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Solving the task requires two different loops:

- In the first one, we change the number that is being printed.
- In the second loop, we take this digit and print it. The number of printouts is the same as the value that is being printed.

This consideration leads to the design of a pair of loops:

```
for i in range(10):  # proceeds from 1 to 9
  for j in range(i): # this line ensures that the printout is
repeated i times
     print(i,end="") # and this printing the value set in the
first loop without breaking a row
  print()            # after printing i numbers, breaks a
row
```

17.1.4

Complete the program that draws a square of stars for the entered **n**.

```
n = int(input('enter the parameter: '))
for i in range(_____):
  for j in range(_____):
     print('*',_____='')
  _____()
```

17.1.5

Write a program that, for entered integer values **m** and **n**, displays **m** rows below each other, with **n** circles (o) in each row.

```
m = int(input('enter the number of rows: '))
n = int(input('enter the number of columns: '))
for i in range(m):
  for j in range(n):
     print('o', end = '')
  print()
```

The solution returns the desired result, but if we look at it in detail, we find that in the inner loop we always execute the same action - 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 - we would print each line in one step.

The modified code would look like this:

```
m = int(input('enter the number of rows: '))
n = int(input('enter the number of columns: '))
# fill the row variable with n characters
row = ''
for i in range(n):
```

```
    row = row + 'o'
# print the entire row m times
for i in range(m):
    print(row)
```

Loops are independent of each other, we may (or may not) use the same control variable.

In the first case, we execute operations in the loop **m** x **n** times, in the second case, we repeat the assignment to a variable **n** times and the printout **m** times - the resulting number of operations is **m+n**.

17.1.6

Complete the program so that it creates a triangle from the characters "x" for the entered **n** as efficiently as possible.

```
x
xx
xxx
xxxx
xxxxx
```

```
n = int(input('enter the parameter: '))
row = _____
for i in range(n):
    row = _____ + '_____'  # add x to the line before the
statement
    print(_____)           # always print the content with one
more 'x' character
```

17.1.7

What will be stored in the **sum** variable after the sequence of commands is executed?

```
sum = 0
for i in range(1,3):
    for j in range(1,4):
        sum = sum + i + j
print(sum)
```

17.1.8

What will be stored in the **line** variable after the sequence of commands is executed?

```
row = ''
for i in range(1,5):
    row = '' + str(i)
    for j in range(1,4):
        row = row + str(j)
print(row)
```

17.2 Searching a string

📖 17.2.1

Despite the fact that we know how to work with a specific string character, we cannot directly change it. So no type assignment is allowed

```
ret  = 'Pokemon'
ret[0] = 'p'
```

The string belongs to the types whose content cannot be changed after assignment. They are so-called **immutable**.

So if we need to change only some characters in the string, we have to create a new one and assign it to the original one, e.g. if in the sentence:

```
sentence = 'My name is ema little.'
```

we need to correct the first and last name, we create a new variable into which we copy the surrounding text and correct the two entered characters, e.g.:

```
sentence = 'My name is ema little.'
sentence1 = sentence[0:11] + 'E' + sentence[12:15] + 'L' +
sentence[16:]
sentence = sentence1
print(sentence)
```

Alternatively, we don't even have to use another variable, but we can insert the result directly into the original variable (sentence):

```
sentence = 'My name is ema little.'
sentence = sentence[0:11] + 'E' + sentence[12:15] + 'L' +
sentence[16:]
print(sentence)
```

📝 17.2.2

Which operations are allowed to work with the variable **ret**.

- ret = 'Hello'
- p = ret[3]
- ret[2] = '-'
- ret[0] = 1
- pom = ret[1] + ret[4]
- ret[1] = ret[2]

17.2.3

Replace the numbers written in words with numbers in the sentence:

```
sentence = 'Three horses with two owners were standing by
house number seven.'
sentence = '_____' + sentence[_____:_____] + '_____' +
sentence[_____:_____] + '_____' + sentence[_____:]
print(sentence)
```

- 2
- 64
- 21
- 59
- 18
- 5
- 7
- 3

17.2.4

The occurrence of a substring in an existing string is verified by the find() function, which returns the position at which the searched substring is located.

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('ga')
print(pos)
```

The **pos** variable will contain the value 4, because at position 4 the beginning of the searched substring was found for the first time.

If the entered substring does not exist in the string, the value -1 is returned. We can use this fact to inform the user.

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('ba')
if pos == -1:
    print('The substring was not found.')
else:
    print('The substring starts at position', pos, '.')
```

17.2.5

What will be the result of the following entry?

```
a = 'Dingo'
b = 'ing'
print(b.find(a))
```

17.2.6

What will be the result of the following entry?

```
a = 'Dingo'
b = 'ing'
print(a.find(b))
```

17.2.7

It is also possible to search the string not from the beginning, but only from the specified position using a variation of **find()** with two parameters, where the second one defines the position from which the search should start.

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('a',10)
print(pos)
```

prints the value 11, which represents the position of the first "a" value from position 10.

17.2.8

What does the following program fragment prints:

```
text = 'Wolfgang Amadeus Mozart'
pos = text.find('g',5)
print(pos)
```

17.2.9

The **replace()** function is also a useful function, which as a result of its operation returns a string in which all occurrences of the first substring are replaced by the second substring. The original string remains unchanged.

It has the form:

```
ret = 'it rained for five days again'
```

```
ret2 = ret.replace('five', '5')
print(ret2)
```

The string **ret** is searched for occurrences of the substring **five** and replaced with the new **5**.

The result is:

```
it rained for five days again
```

17.2.10

What will be stored in the string **ret2** after executing the following commands?

```
ret = 'there is a fly on the wall'
ret2 = ret.replace('e', 'O')
```

17.2.11

Doplň kód tak, aby dával výstup:

```
MaLA has ELA, ELA has LALA
```

```
 ret = 'Mama has Ema, Ema has mama'
ret2 = ret.replace('_____', '_____')
print(ret2)
```

17.2.12

Complete the correct commands/functions to get the desired result:

- string **ret2** should contain 'mama has ema, ema has mama'

```
ret = 'Mama has Emu, Ema has mamu'
ret2 = ret._____
```

- string **ret2** should contain 'MAMA HAS EMA, EMA HAS MAMA'

```
ret = 'Mama has Ema, Ema has mama'
ret2 = ret._____
```

17.3 Working with strings (programs)

⌨ 17.3.1 Sequence of digits

Write a program that prints to the console the number 1 once on the first line, the number 2 twice on the second line, and so on up to 9, the ninth line will have the number 9 nine times in a row.

```
Output:
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

⌨ 17.3.2 A rectangle made of stars

Write a program that, for entered integer values **m** and **n**, will display **m** lines with **n** stars in each line.

```
Input : 2 2
Output:
xx
xx
```

```
Input : 2 5
Output:
xxxxx
xxxxx
```

⌨ 17.3.3 Triangle made of stars

Write a program that reads the number **n** from the user on input and displays 1 star in the first line, two stars in the second line, three stars in the third line ... , **n** stars in the **n**th line.

```
Input : 6
Output:
x
xx
xxx
```

```
xxxx
xxxxx
xxxxxx
```

```
Input : 3
Output:
x
xx
xxx
```

⌨ 17.3.4 Rectangle frame made of stars

Write a program that displays **m** lines with **n** characters to form a rectangle of stars. The inside of the rectangle will be empty, the stars will be only on the perimeter.

At the beginning of the output, do a delineation, i.e. start the printout on a new line. Leave one space at the beginning of the line and one between the stars.

```
Input : 5 5
Output:
 x x x x x
 x       x
 x       x
 x       x
 x x x x x
```

⌨ 17.3.5 Square printout

Write a program that reads the number **n** from the input and prints the numbers from 1 to **n*n** so that in each row and in each column there are exactly **n** numbers that together form a square.

Reserve four spaces for printing each integer variable.

```
Input : 5
Output:
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
  16  17  18  19  20
  21  22  23  24  25
```

⌨ 17.3.6 Compression

Write a program that compresses an entered string of characters. It prints the character first and then the number of occurrences of consecutive characters.

The input is a non-empty string of characters. A list of pairs is printed on the output: a character and a number representing the length of the sequence of its occurrences separated by a colon.

```
Input : 122333444455555444
Output: 1:1 2:2 3:3 4:4 5:5 4:3
```

```
Input : aaaaabbbbbbb  ooo
Output: a:5 b:7  :2 o:3
```

⌨ 17.3.7 Numbers to text

Write a program that replaces all single-digit numbers in a text with a word.

```
Input : I have 1 tent.
Output: I have one tent.
```

```
Input : Divide it by 5.
Output: Divide it by five.
```

```
Input : 1 and 2.
Output: One and two.
```

but:

```
Input : Divide it by 15.
Output: Divide it by 15.
Input : Ta3 is not supposed to be modified.
Output: Ta3 is not supposed to be modified.
```

⌨ 17.3.8 Cancellation of diacritics

Write a program that hides all numerical values in the text with the digit 0.

```
Input : I have 1 tent.
Output: I have 0 tent.
```

```
Input : Divide it by 5.
Output: Divide it by 0.
```

⌨ 17.3.9 The most important word

For the entered text, find out which word appears in it the most times. If there are more such words, write only the one that is closest to the beginning of the text. Pay

attention to characters such as periods and commas, which can distort the result. Pay attention to the case of the letters.

```
Input : Ivan is at home. Ivan bought a new cat. Ivan, be careful.
Output: ivan: 3x
```

```
Input : Divide it by 5.
Output: divide: 1x
```

Functions I.

Chapter 18

18.1 Functions

🕮 18.1.1

The term function

- function is the basic building block of a programming language
- it allows programmers to organize their code into smaller, repeatable, and well-structured chunks

It provides programmers with:

- **modularity** - dividing the code into smaller, simpler and independent modules
- **repeatability** - functions allow repeating tasks to be written only once, while they can then be called multiple times from different places in the program
- **comprehensibility** - functions can be named so that their name describes the activity they provide (including in the programmer's native language)
- **abstraction** - functions make it possible to abstract a specific implementation - other programmers can use the function without needing to know its internal code
- **testability** - functions can be tested separately, which makes it easier to identify and solve errors in the program

📝 18.1.2

Built-in functions

- we already know and use some functions:

```
a = 10.58
b = round(a)
print(b)
c = round(a,1)
print(c)
```

The **round** function provides:

- number rounding independent of the surrounding code (modularity)
- we can call it in the program any number of times (repeatability)
- by the name we can guess what it probably does (understandability)
- we do not know the internal implementation of the function (abstraction)

📝 18.1.3

Complete the names of the corresponding functions to the individual activities:

- generating a random integer in the specified range - _____()

- getting a length of a string - _____()

- obtaining a random number from the interval <0.1) - _____()

- number rounding - _____()

18.1.4

Custom functions

- they perform a simple task or return some value
- they solve a certain partial task
- contains a sequence of commands that are repeated several times in the program
- enables the creation of a program hierarchy, increases clarity
- the function is defined by the keyword **def**
- it is followed by the name of the function, parentheses (with parameters if necessary) and a colon
- the function body is indented similarly to a condition or loop

```
def my_function():
  command1
  command2
  command3
```

In order for the commands placed in the function to be executed, we need to call them somewhere in the code.

```
...
my_function()
....
```

E.g.:

```
def myFunction():
  print("--------------------")
  print("First")
  print("function")
  print("--------------------")
print("Attention, there will be function calling:")
myFunction()
print("End")
```

Without a call in the body of the program, the function code **will not be executed.**

18.1.5

Complete the keywords and function calling

```
_____ myFunction_____
  print("This is")
  print("trial function")

# body of the program
...
_____  # function calling
...
```

18.1.6

How many times is the character "o" printed?

```
def myFunction():
  print("--------------------")
  print("My o function")
  print("--------------------")

print("Attention, there will be a function calling:")
print("End")
```

18.1.7

```
Complete a function called greeting that says hello.
_____ _____():
  print('hello')

_____() # function calling
```

- def
- fun
- greeting
- function
- deff
- greeting

18.1.8

Use of functions

- dividing program into logical modules/units

- it allows us to first prepare the entire program and then formulate its individual parts

```
def introduction():
  print('Hi, I'm a great program')
  print('I count to 10')

def core():
  for i in range(1,11):
    print(i)

def closure():
  print('... and this is the end')

introduction()
core()
closure()
```

18.1.9

Arrange the program so that it prints out the greeting, the introduction, the multiplication table and then the closure:

```
def closure():
  print('End')

def multiplication():
  for i in range(1,11):
    print(i,'*',i,"=",i*i)

def greeting():
  print('Hello')

def whoIAm():
  print('I am a great program')
```

- closure()
- multiplication()
- greeting()
- whoIAm()

18.2 Function with a parameter

18.2.1

Functions with a parameter

We use functions:

- when we want to logically divide the program
- when there is a repeating sequence of statements in a program
- when there is a similar sequence of commands differing only in parameters in the program

Write a program that draws the 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 the specified dimensions, I would first call one, then the other:

```
a = int(input())
b = int(input())
drawRectangle(a,b)
drawRectangle(b,a)
```

- I'm missing just one little thing - a function to draw a rectangle
- In the input there must be the dimensions of the rectangle - parameters
- I will use those as standard variables

```
def draw_rectangle(rows, columns):
  row = 'o' * columns
  for i in range(rows):
    print(row)

# I repeat the body of the program
a = int(input())
b = int(input())
drawRectangle(a,b)
print()
drawRectangle(b,a)
```

18.2.2

Parameters

- the function we used had two parameters

```
def drawRectangle(rows, columns):
  row = 'o' * columns
  for i in range(rows):
    print(row)
```

- 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

```
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 put into the first variable: **4** into the **rows**
- the second to the variable in the second position: **5** to the **columns**

18.2.3

Complete the code for a program that prints two squares on the screen, the first 3x3 and containing the character a, and the second 5x5 made of the character x.

```
def square(a, _____):
  for i in range(_____):
    print(character _____ a)

square(3, '_____')
square(_____, '_____')
```

- *
- character
- a
- x
- 3
- character
- +
- x
- a
- a
- *
- 5

18.2.4

Parameters and arguments

- in general, the function has the form:

```
def function_name(parameter1, parameter2 ... parametern):
  command1
  ...
  commandn
```

- parameters are specified in the function definition, in parentheses after its name, separated by commas
- **argument** is the **value** thatenters the parameters of the function
- if parameters are defined for the function, an argument **must be** entered when calling it

```
def print_text(text):
  print(text)

print_text()
```

Program output:

```
TypeError
vypis_text() missing 1 required positional argument: 'text'
```

18.2.5

Complete the program that will draw a triangle from the characters "x" while the number of characters in the row will be entered at the input.

```
def drawRow(_____)_____
 print('x' * n)

x = _____(input())
for i in range(1, _____):
 _____(i)
```

- n
- n+1
- get
- drawRow
- ;
- :
- x-1
- x

- x+1
- n
- draw_row
- int

18.2.6

Complete the code of the function that prints the sum of the first n natural numbers for parameter n.

```
_____ sum(_____):
    amount = 0
    for i in range(1, n + 1):
        amount _____= i
    print(f"The sum of the first {n} natural numbers is:
{amount}")

x = int(_____())
sum(_____)
```

18.3 Code organization

18.3.1

Functions in different files

- for functions that are not built into the core of Python, we can use the command **import**
- e.g. to calculate the square root, we need the **math** library

```
import math
a = 36
print(math.sqrt(num))
```

- when using a function from another library (file), we include the origin identification (**math**) and the character "." before its name.
- working with random numbers using the **random** library works similarly

```
import random
a = random.randrange(10,20)
print(a)
```

18.3.2

Complete the program that calculates the length of the hypotenuse in a right triangle:

```
_____ _____

a = int(input())
b = int(input())
c = _____._____(a*a + b*b)
print(_____(c,_____))  # rounded to two decimal places
```

- sqrt
- math
- import
- 2
- import
- rand
- 1
- math
- -2
- round
- round
- sqr

18.3.3

Complete the program that prints two random numbers in the range 10-20 and 20-30 including:

```
_____ _____ as rand

a = _____.randrange(10,_____)
b = _____.randrange(_____,_____)
print(a, b)
```

- math
- rand
- 20
- import
- random
- math
- 20
- rand
- math
- 30
- rand

- 31
- 21
- random

18.3.4

Custom functions in separate files

- sometimes it is also convenient to store custom functions in separate files

Let's have a file called **functions.py**

```
# this is a separate file called functions.py
def greeting():
# write the code of your function here
```

- and the file/program that wants to use this feature
- from the **functions.py** file, we can import its functions and use them as follows:

```
import functions

functions.greeting()
```

18.3.5

There is a special.py file containing the functions sum, product and difference.

Complete the code in the my_program file that calls these functions:

```
_____ _____ # provides access to the functions

_____._____    # calling the sum function
_____._____    # calling the product function
_____._____  # calling the difference function
```

- special
- product()
- special
- special
- special
- import
- sum()
- difference()

18.3.6

Selecting functions from a file

- we can also import one separate function from the file
- compared to the previous approach, we get the possibility of calling the function only by name, without the name of the file from which it originates

Let's have a file **functions.py.**

```
# this is a separate file named functions.py
def greeting():
  # ....
```

- and the file/program that wants to use this function
- from the **functions.py** file, we can import its functions and use them as follows:

```
from functions import greeting

greeting() # we call the function directly as if it was a part
of the current file
```

- it is also possible to import several functions at once

```
from functions import greeting1, greeting2
```

18.3.7

A special.py file is defined with the functions sum, product and difference.

Complete the code so that the functions are called as specified in the code. Follow the order in the import as stated in the previous paragraph:

```
_____ special import _____, _____, _____  # provides access to
functions

sum()    # calling the sum function
product()    # calling the product function
_____  # calling the difference function
```

- sum()
- difference()
- difference
- difference()

- sum
- product
- in
- from
- product()
- import

18.4 Functions without parameters (programs)

⌨ 18.4.1 Hello World

Write the code that will use a function without parameters and print a standard programming greeting.

```
Input :
Output: Hello World
```

⌨ 18.4.2 Print numbers

Write the code that will use a function without parameters and print the numbers from 1 to 10 in separate rows.

```
Input :
Output: 1
2
3
4
5
6
7
8
9
10
```

⌨ 18.4.3 Sum of 100

Write the code that will use a function without parameters and print the sum of numbers from 1 to 100.

```
Input :
Output: 5050
```

⌨ 18.4.4 Draw Rectangle

Write the code that will use a function without parameters and print a rectangle of 4x4 consisting of the characters *"o"*.

```
Input :
Output:
oooo
oooo
oooo
oooo
```

⌨ 18.4.5 Draw Triangle

Write the code that will use a function without parameters and print a triangle of height 5 consisting of the characters *"o"*.

```
Input :
Output:
o
oo
ooo
oooo
ooooo
```

⌨ 18.4.6 Draw an Empty Rectangle

Write the code that will use a function without parameters and print an empty rectangle of 10x10 consisting of the characters *"o"*.

```
Input :
Output:
oooooooooo
o        o
o        o
o        o
o        o
o        o
o        o
o        o
o        o
oooooooooo
```

⌨ 18.4.7 Print Name from Input

Write the code that will use a function without parameters and print a greeting for the name given at the input (read the input inside the function).

```
Input : John
Output: Hello John
```

⌨ 18.4.8 Multiplication Table

Write the code that will use a function without parameters and print a multiplication table for the given number (read the input inside the function).

```
Input : 7
Output: 1x7=7
2x7=14
3x7=21
4x7=27
5x7=35
6x7=42
7x7=49
8x7=56
9x7=63
10x7=70
```

18.5 Functions with parameters (programs)

⌨ 18.5.1 Even Digits

Write the code that will use a function with a parameter that contains a number given by the input and print all even digits that are contained in the given number (read the input outside the function and use it as an argument).

```
Input : 123456
Output: 2
4
6
```

⌨ 18.5.2 Odd Digits

Write the code that will use a function with a parameter that contains a number given by the input and print all odd digits that are contained in the given number (read the input outside the function and use it as an argument).

```
Input : 123456
```

```
Output: 1
3
5
```

⌨ 18.5.3 Sequence

Write the code that will use a function with a parameter that contains a number given by the input and print all numbers in the following sequence (read the input outside the function and use it as an argument).

```
Input : 3
Output: 1
12
123
```

```
Input : 5
Output: 1
12
123
1234
12345
```

⌨ 18.5.4 Factorial

Write the code that will use a function with a parameter that contains a number given by the input and print the factorial of the given number (read the input outside the function and use it as an argument). Factorial is calculated following: *n! = 1*2*3*...*(n-2)*(n-1)*n*. The factorial of 0 is 1 and the factorial of a negative number does not exist.

```
Input : 5
Output: 120
```

```
Input : -5
Output: does not exist
```

⌨ 18.5.5 Maximum of three numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the maximum of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3 6
Output: 6
```

```
Input : -5 -11 -7
```

```
Output: -5
```

⌨ 18.5.6 Product of numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the product of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3
Output: 15
```

```
Input : -5 -11
Output: 55
```

⌨ 18.5.7 Difference of numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the difference of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3
Output: 2
```

```
Input : -5 -11
Output: 6
```

⌨ 18.5.8 Rectangle

Write the code that will use a function with a parameter that contains numbers given by the input and print the perimeter and area of the rectangle of the given numbers that are the sides of the rectangle (read the input outside the function and use it as an argument).

```
Input : 5 8
Output: Perimeter is 26 and area is 40
```

```
Input : 15 5
Output: Perimeter is 40 and area is 75
```

⌨ 18.5.9 Triangle

Write the code that will use a function with a parameter that contains numbers given by the input and print the perimeter of the triangle of the given numbers that are the sides of the triangle (read the input outside the function and use it as an argument).

```
Input : 5 8 5
```

```
Output: 18
```

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

⌨ 18.5.10 Surface Area and Volume of a Rectangular Prism

Write the code that will use a function with a parameter that contains numbers given by the input and print the surface area and volume of a rectangular prism from given sides (read the input outside the function and use it as an argument).

```
Input : 5 8 6
Output: The surface area is 236 and volume is 240
```

```
Input : 15 10 12
Output: The surface area is 900 and volume is 1800
```

⌨ 18.5.11 Chessboard

Write the code that will use a function with a parameter that contains string given by the input and prints a grid with the given dimensions *m* x *n* and mark them as a chessboard (*x*, *o*) (read the input outside the function and use it as an argument).

```
Input : 4 4
Output: xoxo
oxox
xoxo
oxox
```

```
Input : 2 3
Output: xox
oxo
```

PRISCILLA

priscilla.fitped.eu