WORK BOOK

Introduction to Python

› **BOOK**
› **EXERCISES**
› **TASKS**

Join the world of programmers

software
development
academy

Introduction to Python

Software Development Academy

Version 2.1, April 2020

Contents

Foreword

Programming is fun – rarely can such words be heard. After all, programming is an innate power enhanced in dark basements illuminated with monitor screens, and is available only to the minds of highly talented adepts of the secret art, preferably before they reach the age of twelve. This skill is basically unattainable for an ordinary mortal. This is a conviction that arose partially due to movies that create the image of a programmer-hacker, an anti-social guy with a long beard, who formulates his thoughts using numbers and who can see colours and sounds in strings of numbers. Partially, it is due to people themselves, who in search of legends and superpowers attribute supernatural properties to programming.

In fact, programming is one of many interesting and very attractive skills that each of us can acquire. You do not need to wade through bloated almanacs or go to Ivy League. You do not need to be born with these skills either. To begin with, this brief discussion will suffice. After reading this Workbook, you will be ready to start writing your first programs yourself.

Summary

This Workbook consists of two parts.

The first part provides a general introduction to programming and discusses the basic structures used in Python. During your grammar lessons at school, you learned about parts of the sentence (subject, predicate, attribute etc.) in your mother tongue, and now you will learn about the basic parts of the sentence (code) in Python:

- simple types and operators
- variables and assignment statement
- conditions
- loops
- complex types

The information contained in Part 1 will be enough for you to write any complicated program. However, such a code will contain many repetitions and probably will not be reader-friendly.

A code is written once, but is read many times. In the second part of our Workbook, you will learn how to avoid repetitions in your code and write a beautiful code that will be a pleasure for other developers to read. The overview covers the two most popular techniques of code refactoring:

- grouping the code into functions
- grouping the code into classes

Each chapter ends with a set of exercises. Exercises are a good way for you to self-check the level of comprehension of the material discussed. In addition, the practical use of the acquired knowledge can significantly help you to fix it in your mind.

All the examples included in this Workbook follow the recommendations of PEP8 (a set of guidelines for formatting a code in Python).

Part I Basics of syntax in Python

Introduction

Programming is the process of giving commands to the computer. Computers do not understand human language. They have their own languages. To give a command that a computer will understand, you must do so in a language that it knows. That is why learning the programming means learning a new language. A language that can be understood by the computer. But programming is something more than just the ability to use another language.

We use a language to express our thoughts. Expressing thoughts in a computer language requires us to slightly convert the way we think. Learning a specific way of thinking is the second step of learning the programming. The way of thinking that is characteristic for programming is often referred to as algorithmic thinking. Although at first you may find it a bit troublesome (algorithmic thinking for most of us is not a natural way of thinking with which were are born), after the appropriate number of exercises you will begin to perceive many issues differently.

Learning the algorithmic thinking is a bit like a good puzzle that requires you to change your mental perspective and look at a particular issue from a slightly different angle. You usually get stuck on such a puzzle, but if you give yourself a little more time for solving it, after a few hours or days you come across a new hint or a new thought that leads you to the solution. You just need to be a bit patient and persistent. A good puzzle is a puzzle where the solution forces you to change your mental perspective. Solving a good puzzle can give you a lot of satisfaction. The same applies to programming. After some time, you will observe with great pleasure how computers begin to obey you and implement your commands with inconceivable speed and precision. However, before this happens you need to get stuck, not give up, and change your mindset a bit. This is a programmer’s everyday life. Let us get started.

Hello world!

Computers, although they often seem to be very smart devices, at the lowest level can only understand zeros and ones. Inside they are just a collection of billions of tiny lights (transistors). For a computer, “one” means that electric current flows through one of these lamps, while “0” means that the current does not flow through this lamp. Computers can turn on and off these lights with the speed of billions of operations per second. They owe all their power to this speed.

The first programs (i.e. sets of instructions for a computer) looked more or less like this:

```
110011100111000001111100000001000011111000011111100000000010000011001111100001
100010000010011111000100000000000001001111100000111110001000000000000000001000
111110010000001100001111100011000000000100111110011100111000111000001000111000
001111100000111110010000011111000110011111100001111000001111000001110011111100
001111000110011100000111000100011111000001111100100000110000000111000001110001
111100011111000111000001000001000011000111110001000001000000011100000111001000
111110001111000001111000011111100001111110000011110000000000000000001111000011
100111000011110011111000111110001111100000100000000000000000000000111110001110
000001110000011100011100111110001000100000000011100001111100110000000010011111
000111100000111100111100010011100000111110000011111001100111100010001111000000
000001000111110010000010011110011001110001000111110001100000100011111000011110
011100111111000111100000111100011111000000011110000011100100001111000100011111
001100011111000111100000111001110001100111100100000000000000111110000011111100
010010000001110000111110010000010001110000011100011001111000100111111000110000
111110001111100011111000001110010000111100010011111000000111110000000011110000011
110000000000000000111000001110000011000001100000111000111000001100111110000111
111001001110000011111000001100011000001001111110000011100110011111000000000111
00000111000011110000110
```

The program directly indicates through which lamps the current should or should not flow. Even if such a record is convenient for a computer, a human needs to work hard to create it. It is even more difficult to guess what such a program does based on the sequence of ones and zeros. Looking at the above “code”, can you guess what this program is supposed to do? It was extremely inconvenient for humans to use such language. That is why other computer programming languages, that were easier to understand for humans, soon began to emerge.

```
org  0x100

mov  dx, msg
mov  ah, 9
int  0x21

mov  ah, 0x4c
int  0x21

msg  db 'Hello, World!', 0x0d, 0x0a, '$'
```

The above code has been written in one of many versions of the Assembler language. Can you guess what the program represented by this code does? Probably you cannot guess it yet, but you can likely recognise some familiar elements in it, such as: “mov” probably means move, and “msg” probably indicates a message.

Along with the emergence of new, more human-readable programming languages, the need arose to translate them into a computer-understandable language, i.e. into a language of zeros and ones. In the IT industry, such a translator from a high-level language (more convenient for humans) to a low-level language (understood by a computer) is called a compiler. Compilers began to emerge, i.e. programs whose task was to translate a program written in the new language into a sequence of zeros and ones.

In the second half of the 20th century, hundreds of new languages and compilers were created. The first version of the Assembler language was created in 1947, in cooperation with John von Neumann (the creator of today’s computers). The Assembler is very hardware-related. Various processors used their own Assembler versions. If you wanted your program to run on several different computers, you needed to write this program in several versions. Therefore, there was a need to write a language whose specification would not change depending on the computer’s parameters. In the early 1970s, Dennis Ritchie developed one of the most popular programming languages in the world – the C language.

```
#include <stdio.h>

int main() {
  printf("Hello, World!");
  return 0;
}
```

You can probably notice here even more elements you can recognise. The meaning of “include” and “main” is obvious, and “printf” probably means “print something”. However, apart from the easily understandable fragments, the C language code still contains a lot of strange characters: semicolons, hash, round brackets, curly brackets and angle brackets. 20 years later, in 1991, Guido van Rossum developed the first Python language specification.

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

The intention of the creators was to develop a programming language with a code that could be read in the manner you read novels. They wanted the Python syntax to be as close to natural language syntax as possible. When looking at the program code written in Python, can you guess what the program does? The chances seem to be much higher than in the case of the previous three versions. And yet each of the four programs presented above does the same: it displays the text “Hello, World!” on the screen.

In the IT, it is a rule that the first program to be written in the newly learned language is the program displaying the words “Hello, World!”.

The first example shows the “Hello, World!” program in the so-called machine language. The second example is the “Hello, World!” program written in the Assembler language for 64-bit Intel processors. The third example is the “Hello, World!” program written in the C. Python, which is even a higher-level language. The Python translator for the machine language that we will be using here (CPython) was written in the C language. The creators, while writing the Python language specification, were mainly inspired by the ABC, Smalltalk and ALOGL 68 languages. Programs that translate a code into a series of zeros and ones in languages such as Python are not called compilers, but interpreters. At the moment, the differences between compilers and interpreters are not important to us.

The “Hello, World!” program performs the function of an initial test. In this way you can verify whether you have installed and configured everything correctly, and whether you can run the program you have written. Although programs can be written in any text editor, we will use the fully-configured and ready-to-use Repl.It integrated development environment. It is available for writing programs at https://repl.it/languages/python3.

In addition, Appendix A provides complete installation and configuration instructions for the PyCharm integrated development environment. PyCharm is the integrated development environment recommended by the SDA. Read Appendix A if you want to run these examples regardless of whether or not you have access to the Internet.

Let us start with the “Hello, World!” program. Go to https://repl.it/languages/python3 and in the middle window type in:

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

Then click the Run button at the top of the page. After a while, the result of your program will be displayed on the right.

Have you managed to run this program? Congratulations! You have just written your first program in Python. Your program displays the text “Hello, World!” on the screen.

Exercises

1. If you have not done it yet, try to write and run the “Hello, World!” program in one of the Integrated Development Environments (IDE). In Appendix A you will find instructions on how to install and run the PyCharm integrated development environment
2. Write a program that displays "Hello, <Your name>!" on the screen (instead of “<Your name>”, enter your name).
3. Remove one of the quotation marks around the 'Hello, World’ text. Try to run the program. Read the message describing the error that has occurred. Then correct the quotes and try to spoil the program in a different way (each time read the description of the error encountered), e.g.:
   - remove or add a bracket
   - remove one of the letters in the word “print”
4. Before executing the following lines of the code, try to guess what the result will be:
   - print("What a beautiful day")
   - print("What", “a” "beautiful", "day")
   - print(3)

    - print(3, 4, 5)
    - print("1", "2", 3, 4, 5)
    - print("What", “a” "beautiful", "day", sep=",")
5. Write a code line starting with #. Executing such a code line will not cause any action. In Python, the # character is used to comment on the line. When the Python interpreter encounters the # character, it goes to the next line of the code. Therefore, this character is used to enter comments in the code. Comments will be useful for other programmers who read your code, or for yourself.

Basic features and philosophy of Python

Python is a general-purpose (multi-paradigm) language. This means that it does not force the programmer to use a particular way (paradigm) to write the code. In Python, you can easily group your code into functions or classes, or in many other ways. It has an extensive standard library, which includes, among others:

- Math – a library for mathematical calculations (it has such functions as evolution, sin, cos, …)
- Datetime – a library to work with data representing time (hours, days, months, years, …)
- Collections – a library to work with custom data structures

In addition, it contains many external libraries. Python has achieved the greatest popularity due to its libraries designed for:

- network programming (Django, Flask, Pyramid)
- machine learning and artificial neural networks (scikit-learn, PyTorch, Keras)
- scientific calculations (Numpy, Matplotlib, Scipy, Pandas, TensorFlow)

This is just a small part of the huge collection of Python libraries available in the Internet. There you can find libraries for writing graphical user interfaces (PyQt, Tkinter, WxPython), libraries for writing games (Arcade, Pygame), libraries for working with sound (playsound, simpleaudio, pyaudio) and many others.

Python is an interpretable language. This means that when running a code written in Python, you do not need to go through the compilation phase of your code on your own. You simply run the file with your code and Python does the rest for you.

The Python developers have gathered 19 of the most important principles that guided their language creation into one collection called the Zen of Python. To display its content, enter the following text in the middle window:

```
import this
```

and click the Run button.

In the window on the right, you will see the Python manifesto,

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The above rules are a set of 19 disciplines that you should follow while writing your code. Let us look at principle No. 6. At first, its content may seem to be the strangest one.

```
Sparse is better than dense.
```

Just as a natural language allows expressing the same thought in many ways, in a programming language there are many ways to write down the same operation.

When you are working on any issue, it gets closer to you over time. After a few days of work, you may be tempted to present all collected conclusions in the most concise form possible. The fifth version of our code may delight us with its content. We have managed to pack in one line the issue which was initially written in 12 lines.

```
print('\n'.join(f"{j} bytes = {j*8} bits which has {256**j-1} possible values." for j
in (1 << i for i in range(8))))
```

We have made our code as dense as possible. It is very concise. However, there is a significant risk that people who were not co-working with us on this code and who will have to read it will not be so much delighted. We can write the same in a few lines:

```
for i in range(8):
    j = 1 << i
    print(f"{j} bytes = {j*8} bits which has {256**j-1} possible values."
```

The result is the same, but the code is much clearer now. Principle No. 6 recommends avoiding the overloading of code with content at the expense of its readability. As you become more experienced, the principles contained in the Python manifesto will become increasingly understandable to you.

Repl.It - development environment

Before we start writing other programs, let us get to know our development environment a little better.

To write and run Python programs, all we need is a computer and an access to the Internet. If you would like to run your programs regardless of whether or not you have Internet access, you will need an integrated development environment. In Appendix A you will find the instructions for installing and configuring PyCharm – the integrated development environment recommended by the SDA.

When you enter the Repl.It site, you will see three windows.

The first window (on the left) contains a preview of your project directory. It is where you will create your new files and folders. When the volume of your code increases, you will need to split it into files and, over time, even into folders. You do it in the first window. On startup, Repl.It automatically creates an empty main.py file in the project directory. Python code files are called scripts. The convention requires that the scripts be given the py extension. The second window, in the middle, displays the content of the file indicated in the project directory (in our case, it is main.py). For now it is an empty file. In the previous chapter we wrote the following code in it:

```
"Hello, world!"
```

and then we executed the script by clicking the Run button. After clicking the Run button, the Python interpreter:

- started operation
- translated (interpreted) our script (print ("Hello, World!")) into zeros and ones
- forwarded zeros and ones to the processor
- received the output from the processor
- displayed the output received from the processor in the third (right) window,
- finished operation.

In this way, we have started the Python interpreter in the script mode.

The Python interpreter can also operate in another mode, i.e. the interactive mode. The interactive mode is often referred to as the Python Shell. The third window (on the right) is responsible for the interactive mode in the Repl.It environment. In the script mode, the third window is a table where the program execution result is displayed. In the interactive mode, you can use the third window for both writing the code and displaying the result. In the interactive mode, enter your code in the right window and press Enter to make the result displayed below. This mode of operation is also known as the REPL. Each letter of this acronym corresponds to the relevant operation that the Python Shell performs when Enter is pressed. And so:

```
R (Read) means reading
E (Eval) means evaluating (calculating)
P (Print) means displaying
L (Loop) means the return to the initial state.
```

Let us trace how the Python Shell works, using the “Hello, World!” program as our example. After starting the Repl.It environment, the > character appears in the third window. This is the so-called prompt sign that indicates the place to enter the code. The Python Shell is in the waiting mode.

After you enter the following text in the right window:

```
"Hello, world!"
```

and press Enter, the Python Shell will:

- Read ® your input
- Evaluate (E) your input (in this case, there is nothing to evaluate)
- Print (P), i.e. display the output on the screen
- Loop (L), i.e. return to the waiting state.

The output has been displayed in the right window and the prompt sign below indicated that the Shell returned to the waiting state.

Please note that the above code line, while run in the script mode (the middle window), will not return any output (the print function is missing). This did not happen in the interactive mode, because the automatic display of the output (P in the REPL acronym) is one of the stages in the interactive mode. You do not need to remember to insert the Print statement, because the Python Shell automatically displays the output. While adding the Print statement in the script mode changes a lot, it does not matter at all in the interactive mode.

The interactive mode is used to present or test small pieces of code at a time. A code entered in the Python Shell cannot be saved. When the code becomes multi-line, the script mode is no longer convenient. For the sake of clarity of the issues being introduced, at the beginning we will be working in the interactive mode. Over time, when our code ceases to be one-line, we will start to work in the script mode.

Before moving on to new issues, let us review the “Hello, World!” program again.

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

You can see the print statement and the opening bracket here. Then there is a quotation mark and a string of a dozen or so characters, followed by the second quotation mark and the closing bracket. In Python, expressions such as print are referred to as functions. Our function is called print. The task of the print function is to display the parameters passed to it. The parameters that we pass to the function are placed inside the brackets, after the function name. In this case, the parameter that we pass to the print function is "Hello, World!” (strings are placed in quotation marks). The print function displays the parameter passed inside the brackets on the computer screen. In our case it is the "Hello, World!” text. We have just begun analysing Python’s grammar (or more specifically, its section called syntax). You did the same in elementary school during your native language classes. During your grammar course, you learned about parts of the sentence, such as subject and predicate. Here we will have types and operators, assignment statements, conditions, loops, functions and classes. Let us take a closer look at the first of them, i.e. types and operators.

Simple types and operators

We already know how to use Python to write a program that displays a text on the screen. It is a good start. But how to make Python do all these amazing things that modern computers can do? To learn this, you need to learn a little more about Python itself.

First, let us see whether Python can display more than just strings. Can Python display numbers?

Type in 3 in the right window and press Enter. You will see number three in the line below.

```
> 3
3
```

Here is what the Python Shell has done after Enter was pressed:

1. R – it has read what you had typed in
2. E – it has evaluated it (in this case the calculation was trivial again)
3. P – it has displayed the output
4. L – it has returned to the initial state and displayed the prompt sign.

And can Python sum up numbers?

```
> 3 + 5
8
```

It can.

What about multiplying?

```
> 3 * 5
15
```

And dividing?

```
> 3 / 2
1.5
```

Now let us write a slightly more complex expression, e.g.

```
> 3 * (2 + 4) - 3
15
```

It works!

Let us try once again. Here is what the Python Shell has done after Enter was pressed:

1. it has read your code (Read)
2. it has evaluated the result (3 * (2 + 4) – 3 = 15) (Eval)
3. it has displayed the result on the screen (Print)
4. it has returned to the waiting state (Loop).

This means Python can add, subtract and multiply numbers and it can understand parenthesis notation. Can Python add strings? Let us check it:

```
> "Hello world" + "!"
Hello world!
```

It can. The operation of adding strings is called concatenation. Can it multiply them?

```
> "Hello world" * "!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

We have received a weird error. Errors are Python's way of communicating that something went wrong. As you see, Python cannot multiply strings.

What we have just started to do is the examining of the different types of data that exist in Python (for now we only know the strings and numbers) and the operations allowed on these data (for now we have learned the addition, subtraction, multiplication and division for numbers and concatenations for strings).

There are more types in Python. The basic division classifies types into simple and complex ones. Complex types will be discussed later in this Workbook. The simple types include:

- integer
- float
- string
- bool
- None

If you want to check the type of the entered value, you can use the type function for this. Just as the print function is used to display on the screen the value passed inside brackets, the type function is used to display the type of this value.

Let us check the type of the value of 3.

```
> type(3)
<class 'int'>
```

Python has displayed <class 'int'>. For now, we will not be dealing with the word class. Beside it, we can also see: int. Int is an abbreviation for integer, which means an integer type. It is one of the three numeric types supported in Python.

The most popular numeric types in Python include:

- integer type
- floating point type

Integer type

The integer type represents integers. The values that belong to the integer type are e.g.

```
> 3
> 24562
> -5364
```

Values acceptable for a given type are often referred to as literals. You can consider the above values to be examples of valid integer literals.

The integer type supports all basic mathematical operators, such as:

- Adding
- Subtraction –
- Multiplication *
- Division /
- Exponentiation **
- Integer division //
- Remainder from division (modulo) %

Floating point type

The floating point type represents real numbers. Examples of floating point literals are as follows:

```
> 3.4
> 4.25
> -6.32314
> .4
```

Numeric types can be compared to each other:

```
> 3 == 3.0
True
> 4 == 2
False
> 5 < 3
False
> 45.3 >= 10
True
```

The relational operator group includes the following:

- Equal to ==
- Not equal to !=
- Greater than >
- Greater than or equal to >=
- Less than <
- Less than or equal to ⇐

The relational operator calculates its right-hand expression, then calculates its left-hand expression, and returns True if both of the received values are equal; otherwise it returns False. One of two boolean type values. And what is the boolean type?

Boolean type (bool)

The boolean type represents a logical value in Python. It takes one of the two values: True or False. Boolean literals are as follows:

```
> True
True
> False
False
```

Boolean types work on all mathematical operators. In mathematical operations, True is treated as 1 and False as 0.

```
> 3 + True
4
> 2 * False
0
> 5 / True
5.0
```

In addition, boolean types support three logical operators that we know from a basic course of logics: and, or and not.

The and operator corresponds to a logical conjunction. Conjunction is only true when both arguments are true. It corresponds to the conjunction “and” in a sentence.

```
> True and True
True
> 5 > 3 and 1 < 3
True
> True and False
False
> 5 > 3 and 1 > 3
False
```

The truth table for conjunction is as follows:

p	**q**	**p and q**
1	1	1
1	0	0
0	1	0
0	0	0

The or operator is a logical alternative. Alternative is not true only when both arguments are not true. It corresponds to the conjunction “or” in a sentence.

```
> True or False
True
> 5 > 3 or 1 > 3
True
> False or False
False
> 5 < 3 or 1 > 3
False
```

The truth table for alternative is as follows:

p	**q**	**p or q**
1	1	1
1	0	1
0	1	1
0	0	0

The not operator corresponds to a logical negation. Negation is a unary action. It returns a negation of an argument.

```
> not True
False
> not 5 > 3
False
> not False
True
> not 1 > 3
True
```

The truth table for negation is as follows:

p	not p
1	1
0	1

String type

The string type represents strings in Python. A string is created by enclosing the string of characters in single or double quotes. Examples of string literals:

```
> 'John likes Mary'
John likes Mary

> "John likes Mary"
John likes Mary
```

In this way you can create one-line strings. If you want to create a multi-line string, you should use three apostrophes.

```
> """John likes Mary,
... Mary likes John"""
'John likes Mary,\nMary likes John'
```

The \n character in the output line is the Python's way to remember the location of the newline character.

Strings can be added:

```
> "John " + "likes " + "Mary" + "."
John likes Mary.
```

The operation of adding strings is called concatenation. In addition, strings can be multiplied by integers. This operation is called a multiplication of the string. The remaining mathematical operators do not work on strings.

Strings can be compared.

Strings are compared alphabetically, letter by letter. Lowercase letters are larger than uppercase letters. Space or no character is always smaller than any other character.

```
> ‘John' == "John"
True
> 'John' == 'john'
False
> "john" > "John"
True
> "john" > "john likes"
False
```

None type

In Python, the None type represents the lack of value. It is usually used to indicate the lack of data. A literal of the None type is as follows:

```
> None
```

Appendix B contains the list of all basic Python types and the list of all operators available in Python.

Casting – conversion of types

Sometimes you may want to change the type of the value already entered. Changing the value type is called conversion or casting. This is done with the use of casting functions. Each type (except the None type) has its casting function:

- the function casting to the integer type: int ()
- the function casting to the floating point type: float()
- the function casting to the string type: str()
- the function casting to the boolean type: bool()

If you want to cast the integer type (e.g. 5) to the floating point type, just enter:

```
> float(5)
5.0
```

In this way, from the integer type (literal 5) you have obtained the floating point type (literal 5.0). Casting in the opposite direction (from the floating point type to the integer type) rounds down the value:

```
> int(4.2)
4
> int(4.9)
4
```

Not all casts are possible. If you try to cast the string type value to the integer type, you may receive an error.

```
> int("1024")
1024
int> “John likes Mary")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'John likes Mary'
```

If the string of characters between quotation marks does not create an integer literal, attempting to convert it to the integer type will result in an error. In our example, Python knows how to convert the string of characters "1024" to an integer, but it does not know how to convert the string "John likes Mary" to an integer. It informs us about it by throwing an error. When an error occurs, Python terminates the program and displays information about the error. In our case, it is a ValueError: an invalid literal passed to the int cast function: 'John likes Mary'.

Comments

Comments are used to place information in the code for people who will read it. Comments are created using the # symbol.

```
> int("1024")  # Watch out for casting wrong literals
1024
```

When the Python interpreter encounters the # character, it ignores all subsequent characters and moves to the next line.

Exercises

1. Determine the type of the following literals (values):

```
"Hello"
2
2.
2.0
"2"
4.55
-4.3
54 + 5j
True
None
```

2. Determine the type of the following expression results:

```
2 + 2
2 + 2.
4 / 2
4 / 3
2. * 4
3 * "a"
```

3. Calculate the arithmetic mean of any five numbers.

4. Calculate the following expressions:

```
3 + 45 - 2.5 * 4
2 ^ 3
3.5 * (3-2)
501.0 - 99.9999
10.0 / 4.0
10.0 // 4.0
5 % 2
2 == 2
1 != 1
99 > 1.1
True or False
2 == 2 or 1 != 1
True and True and False
2 == 2 and 99 > 1.1 and 1 != 1
not False
not 1 != 1
```

5. Correct the logical expressions so that each expression returns the True value.

```
"asd" == "qwe"
3>6
"as" <= "ad"
"z" < "n"
45 != 45
```

6. Perform the casting as follows:
   - cast the following values to the integer type: 15.5, 4.01, 4.99, -4.01, -4.99
   - cast the following numbers to the floating point type: 0, 100, -4
   - cast the following numbers to the boolean type: 2, -6.1, 'a', "abc", 0, 0.0, "", " "

Please note that the only values that the bool operator casts to False are 0, 0.0 and an empty string. All the remaining values are cast to True. Values that after casting to bool have the False value are called Falses, and those that have the True value are referred to as Truths.

Variables and assignment statement

The issues discussed to this point allow us to perform single calculations. The result is displayed on the screen. But what if you want to use the obtained result for further calculations? You need something where you can store your result. You cannot achieve this with types and operators alone. You need a new syntactic element.

It would be convenient to give your value some name (e.g. My_value), so whenever you want to use this value, you can just call it by name (my_value). In programming, this function is performed by variables. A variable is a name that indicates the value. You can picture it as a box with a label. The label contains the name of the box. You can put any value into the box.

```
> a = 4
>
```

In the example above, we have put the value of 4 into “the box labelled as a”. We say that the variable a is assigned with the value of 4.

```
> a
4
```

This time, we “take” the contents “out of the box” labelled as a. We say that we return the value **stored** in the **variable** a.

The name assigned to a variable is often referred to as the identifier. Identifiers can be of any length. Python allows identifiers to be composed of letters, numbers and the underscore character (_). Python distinguishes between uppercase and lowercase letters, so identifiers that differ only in letter case are different (my_value is a different variable than My_value). Identifier must not start with a number. Besides, it cannot be any of 33 keywords (reserved), e.g. def, class. A table containing the list of Python keywords can be found in Appendix B.

The = sign is called the assignment operator. Please do not confuse it with the equation mark that you know from your maths lessons. The operation of the assignment operator boils down to calculating what is on the right side of the operator and assigning the result to the identifier (name) on the left side of the operator. The assignment operator together with its left- and right-hand sides is called an assignment statement. Examples of assignment statements:

```
> a = 3
```

The result of the right-hand calculations (3) is assigned to the variable a.

```
> b = 4 * 2 + 4
```

The calculation result on the right (12) is assigned to the variable b. The variable can store data of any type:

```
> my_value = 3
> my_string = "asd"
> x = True
```

You can put variables into the function

```
> a = 2.5
> print(a)
2.5
> type(a)
float
> int(a)
2
```

You can perform operations on variables that are characteristic for the stored type:

Example 1

```
> a = 4
> b = 5
> c = a * b
```

Example 2

```
> d = "john likes "
> e = "mary."
> f = d + e
```

Remember: The = operator has a different function in programming than in mathematics. The = operator in mathematics indicates equality. Equality is a statement saying that what is on the right is equal to what is on the left. In programming, the = operator means assignment. Assignment means calculating of what is on the right side of the operator and saving the result under the name on the left side of the operator. In particular, lines such as:

```
> a = 4
> a = a + 2
```

make no sense in mathematics. The second line contains a false statement (4 = 4 + 2). In programming, on the other hand, the second line means: calculate what is on the right (4 + 2, i.e. 6) and assign the result to the left side (i.e. to the variable a). Before the line (2) execution, the value of the variable a was 4. After the execution of this line, the value of a will be 6.

Since the = sign in Python is an assignment operator, is there any equivalent of the mathematical equality mark in Python? The equivalent of the mathematical equality operator in programming is one of the relational operators discussed in Chapter 3, i.e. equality operator ==. In addition to the equality operator ==, Python has another operator that performs a similar role, i.e. the identity operator is.

The difference in operation between these two operators is well illustrated by their names. The equality operator checks whether the values of the compared types are equal. In particular, an integer type with the value of 4 is equal to a floating point type with the value of 4.0.

```
> 4 == 4.0
True
```

However, these are not the same, identical values (they are of different types). You can verify this using the identity operator is.

```
> 4 is 4.0
False
```

Exercises

1. Create variables belonging to the following types:
    - integer
    - floating point
    - string
    - bool
2. Create the person_age variable and assign an integer type value to it. Display the value and type of the age_person variable on the screen
3. Cast the floating point variable with the value of 4.3 into the integer type. Cast the string type variable with the value of "45” into the integer type.
4. Write a program that displays the string "Hello, <Your name>! on the screen. Make your name stored in a string type variable..
5. Declare several variables of different types and test the operation of the following operators on them:

    ```
    -
    *
    /
    **
    ```

6. Declare two variables corresponding to the sides of the rectangle and count its area and perimeter. Formula for:
    - area of a rectangle: P = a * b
    - perimeter of a rectangle: L = 2*a + 2*b

7. Declare the variable corresponding to the diameter of the circle and calculate its area and circumference. The relevant formulas are as follows:
    - circle area P = pi * r^2
    - circle circumference L = 2 * pi * r

Assume 3.14 as the value of pi.

Conditions

The programs we have written so far performed the same set of statements each time. Regardless of the data you enter into such a program, an identical, linear sequence of commands will be executed each time (i.e., do something, remember something, write something on the screen).

```
> a = 7
> b = 5
> c = 9 / (a-b)
> int(c)
4
```

What if in Exercise 2 from the previous chapter we would like the program to print the message "person is adult" on the screen instead of the stored age, if the value of the age_person variable is greater than or equal to 18? You will not achieve this by using only types, operators and variables.

For such a task you will need another syntactic element. Our program is supposed to behave differently depending on what value the age_person variable takes. The statement that allows you to "branch out" the control flow is called a condition.

```
age = 20
if age  >= 18:
    print("Adult")
```

We have used the if statement here. The syntax of the if statement can be schematically represented as follows:

```
if boolean expression:
    statement1
    statement2
```

The if word is followed by a boolean expression (True, False) and a colon. The if word together with the boolean expression that follows it form a condition. Below, indented, place a block of the code to be executed if the boolean expression contained in the condition is true (i.e. returns True).

Note the indentation in the code. In many languages, braces are used to stand out the code block. In Python, indentation is used for this. That is why whitespace is so important in Python. Placing a statement at the wrong indentation level will cause the program to malfunction. In special cases it will result in an error. PEP8 recommends that indents in the code be entered using four spaces.

And what happens if the condition is not met? The Python interpreter will skip the code lines inside the condition, go further, reach the end of the script and finish operation. To support a case when the condition is not met, the if / else statement is used.

```
age = 20
if age  >= 18:
    print("Adult")
else:
    print("Minor")
```

The if / else conditional statement can be presented using the following scheme:

```
if boolean expression:
    statement1
    statement2
else:
    statement3
    statement4
```

The code block under the else statement will be executed **if** the condition is not met (i.e. it returns False).

To put more than one condition in the code (more than one branch), the elif statement (short for elseif) is used. Suppose you want your program to print:

- "adult” – if the value of the age variable is greater than or equal to 18
- "adolescent” – if the value of the variable age is within the range (16, 18)
- • "minor” – otherwise

```
age = 20
if age >= 18:
    print("Adult")
elif age >= 16 and age < 18
    print("Adolescent")
else:
    print("Minor")
```

Further conditions can be added by duplicating the elif statement. The if-elif-else sequence is a complete image of the conditional statement.

Exercises

1. Write a program that will display on the screen whether the value of the variable is divisible by 7
2. Write a program that will display the text “The value of the variable is odd” on the screen if the value of the variable is odd, and otherwise display the text “The specified value is even”.
3. Write a program that will display on the screen whether the variable value is divisible by 7, by 5 or by 3. If the variable value is not divisible by any of these numbers, the program should display an appropriate message.

4. Write a program that will display the following results for the specified variable:
    - the difference between the value of this variable and 17 **if** the difference is less than 17
    - o the square of this difference, otherwise.
5. Write a program that returns the sum of the three numbers given. **If** all three numbers are equal, the program will return the triple value of their sum..

Loops

So far, when we were running the program, each of the code lines we wrote was executed exactly once. However, when we run a good game, we can play it for several hours or even days. Does this mean that the code of this game contains a huge number of lines and for several days the computer goes on each of these lines only once? No, but you would not achieve that effect with the types, operators, assignment instructions and conditions alone. You need another syntactic element.

Our game uses a structure that in programming is called a loop. A loop is an instruction to repeat the indicated code block.

There are two types of loops in Python:

- while loop
- for loop

While loop

The while loop is also called a conditional loop. It executes the code block it contains as long as the boolean expression within the loop is met.

```
x = 0 ①
while x < 5: ②
    x = x + 1 ③
    print(x) ④
```

① The 0 value is assigned to the x variable.

② The loop condition meaning: do the following as long as x < 5

③ The first line of the loop body. Increases the x value by one. After the execution of this line, the value of x will be 1.

④ The second line of the loop body. Prints the current value of x on the screen. The current value of the variable x is 1.

At the end of the loop body, the Python interpreter returns to the loop condition. It checks whether the loop condition is still met (1 < 5). If the condition is met, it executes the loop body again. After the next body execution, the value of x will be 2. The sequence is repeated as long as the loop condition is met, i.e. as long as the value of the variable x is less than 5. The first value of the variable x that will not meet the condition of the loop is 5. The x reaches the value of 5 in the fifth loop. The Python interpreter will print the value on the screen, and then once again go to the loop condition which this time will not be met. At this point, the Python interpreter will terminate the loop execution because the loop condition is no longer met.

The while loop can be represented by the following scheme:

```
while boolean expression:
    statement1
    statement2
```

The while loop consists of a condition and a body. The word “while” followed by a boolean expression and a colon is called a loop condition. The body of the loop is the statement block placed directly below it. The loop body ends with a statement with the same or lower indentation level as the loop condition, and in the absence of such a statement, with the end of the file. The loop body execution is repeated as long as the loop condition is met. Note! If the loop condition is always met, an infinite loop is received. In particular, if you write:

```
while True:
    print("Hi")
```

the Python interpreter will start to display Hi on the screen again and again. An infinite loop can be interrupted with a shortcut Ctrl+C (the so-called break sign).

For loops will be discussed in the next chapter, just after discussing complex types.

Exercises

1. Write a program that displays all natural numbers between 0 and 50.
2. Write a program that displays all even numbers between 0 and 100.
3. Write a program that displays the squares of all integers between 0 and 10.
4. Using the loop, write numbers from -20 to 20. Then write:
    - the first 6 numbers
    - the last 6 numbers
    - all even numbers
    - all numbers except digit 5
    - all numbers up to and including digit 7
    - all numbers divisible by 3
    - the sum of all numbers
    - the sum of numbers greater than or equal to 4
    - all numbers and their powers
    - all numbers and their values for modulo 10
5. Write a program that displays numbers that are multiples of 5 and are divisible by 7 within the range from 1500 to 2700.
6. Write a program that writes numbers from 0 to 6 and omits 3 and 6. Do it in two versions: with and without the continue statement.

Complex types

In addition to the basic types discussed in Chapter Three, there are also complex types in Python. Complex types are containers for simple types. You can store many values in them. The basic complex types in Python include:

1. List (ang. list)
2. Tuple (ang. tuple)
3. Dict (ang. dict)
4. Set (ang. set)

The list and set are the most commonly used complex types in Python.

List

A list is a container in which you can place any number of values of any type. Examples of list literals are as follows:

```
> []  # Empty list
[]
> [1]  # One-item list
[1]
> [1, 4]  # Two-item list
[1, 4]
> [1, 4, "John likes Mary", -4.5]  # Multi-item list
[1, 4, "John likes Mary", -4.5]
```

A list can be assigned to a variable:

```
> a_list = [1, 3.4, 0]
```

You can add two lists together (concatenation of lists):

```
> list_1 = [1, 3.4]
> list_2 = [-4.5, 0]
> list_1 + list_2
[1, 3.4, -4.5, 0]
```

A list can be multiplied by a number (multiplication of the list):

```
> list_1 = [1, 2, 3]
> list_1 * 2
[1, 2, 3, 1, 2, 3]
```

Using the len function, you can check the length of the list:

```
> a_list = [0, 4.5, 2.32, -5]
> len(a_list)
4
```

Indexes provide access to individual items of the list. The first item of the list has index 0. The next item has index 1. The index of the last item of the list is the length of the list minus 1. Individual indexes of the list are referred to using square brackets.

```
> a_list = [1, 3.4, -5, 0, 3, 4, 5]
> a_list[0] # first item of the list
1
> a_list[1] # second item of the list
3.4
> a_list[len(a_list)-1] # last item of the list
5
```

Note that the **first** item of the list has index 0. The second one has index 1. This is a common cause of errors. Lists in Python are indexed from 0.

Python also supports inverse indexes. The last item of the list has index -1, next to last has index -2, and the first one has index -length of the list.

```
> a_list = [1, 3.4, -5, 0, 3, 4, 5]
> a_list[-1]
5
> a_list[-2]
4
> a_list[-len(a_list)]
1
```

Fun fact

Python treats strings as a special type of list consisting of single characters. Strings support all of the operations on lists discussed above.

You can check the length of the string with the len function:

```
> a_string = "John likes Mary"
> len(a_string)
11
```

The individual characters of the string can be accessed using index notation.

```
> a_string = "John likes Mary"
> a_string[0]
'A'
> a_string[4]
'm'
> a_string[-1]
'a'
```

Dictionary is another complex type frequently used in Python.

Dictionary

Dictionary is not an indexable type, i.e. it has no indexes. Its name reflects its character very well. Just as in language dictionaries you can find the term and its definition, in the dictionary type you have the key and the value. A restaurant menu has a dictionary structure where the names of the dishes are the keys, and the values are their prices. A phone book has a dictionary structure where the keys are the names of the people, and the values are their phone numbers. A dictionary stores key-value pairs. We can say that a list is a special case of the dictionary, where the keys are consecutive natural numbers (indexes).

Examples of dictionary literals are as follows:

```
> {}  # Empty dictionary
{}
> {'key1': 1}  # One-item dictionary
{'key1': 1}
> {'key_1': 1, 'key_2': 4}  # Two-item dictionary
{'key_1': 1, 'key_2': 4}
> {'a': 1, 'b': 4, 'c': "John likes Mary", 'd': -4.5}  # Multi-item dictionary
{'a': 1, 'b': 4, 'c': 'John likes Mary', 'd': -4.5}
```

Remember!

A key in the dictionary must have a unique value. This differs the dictionary type from the address book in which you can find dozens of John Smith names.

To refer to individual items of the dictionary, use the key.

```
> a_dict = {'a': 1, 'b': 4, 'c': John likes Mary", 'd': -4.5}
> a_dict['a']
1
> a_dict['c']
'John likes Mary'
```

For loop

The for loop, like any other loop, is used to repeat a block of statements, but we use it in a slightly different way. The for loop passes through subsequent elements of the complex type, each time executing the statements contained in its body.

```
a_list = [1, 3.4, -5, 0] ①
for item in a_list: ②
    print(item) ③

1
3.4
-5
0
```

① A 4-item list is assigned to the a_list variable.

② We begin going through the items of our four-item list. Inside the body, we will refer to the next elements of the loop using an identifier.

③ The loop body. We print the current item of the list on the screen.

The for loop is the so-called syntactic sugar. It does not introduce anything new to our substantial collection of development tools. The same effect can be obtained by using the previously learned syntactic elements.

```
a_list = [1, 3.4, -5, 0]
idx = 0
while idx < len(a_list):
    item = a_list[idx]
    print(item)
    idx = idx + 1

1
3.4
-5
0
```

However, thanks to the introduction of the for loop, our code is clearer and more concise. The operation of going through all the items of a collection is called iterating by its items. With the for loop, you can easily iterate over the items of the collection.

You can iterate by the characters of the string:

```
a_string = "John likes Mary"
for char in a_string:
    print(char)

J
o
h

n
l

i
k
e
M
a
r
y
```

You can iterate by dictionary keys.

```
a_dict = {'a': 1, 'b': 4, 'c': "John likes Mary", 'd': -4.5}
for item in a_dict:
    print(item)


'a'
'b'
'c'
'd'
```

Summary of Part I

Here we have finished discussing the basic syntactic elements of Python. The information presented in the first part allows you to write any complex program. Any existing computer program, game or web portal can be written using the same types, operators, assignment statements, conditions and loops.

Exercises

1. Write a program that will display, one by one, all items together with their types for a given list. For the following list:

```
a_list = [4, True, None]
```

the program should display the following result:

```
4 <class 'int'>
True <class 'bool'>
None <class 'NoneType'>
```

1. A 10-element array is given: a_list = [1, 3, 5, 2, 5, 6, 7, 4, 9, 7]. Write a program that prints:
   - all digits,
   - the first 6 digits,
   - ostatnich cyfr,
   - all even digits,
   - all digits on odd indexes,
   - all digits except digit 5,
   - all digits up to and including digit 7,
   - all digits divisible by 3,
   - the sum of all digits,
   - the sum of digits greater than or equal to 4,
   - the smallest and the largest digit.
2. Write a program that for the given number of words, e.g. a_list = ['cat', 'primer', 'window', 'computer'] will display subsequent items of the list together with information about the length of these items.
3. Write a program that for the given list of words, e.g.

```
list_of_words = ["spam", "table", "spam", "brown", "air", "maleware", "spam", "end"]
```

will display only the list items that have no "spam" value. In addition, **if** the list item value is "maleware", the program should terminate operation immediately. Use the break statement. Search in the Internet for information on how the break statement works.

Part II Code refactoring

Introduction

Assignment statement, condition and loop are enough for any application or game to be written. At this point you may not see it yet, but you can write any complex program using the techniques you have learned. You just need some practice for this

Now when you know how to write programs, it is time to learn how to write them perfectly. A program written only with assignment statements, conditions and loops will work, but its code will be difficult to read and develop. It will contain many repetitions of the same lines of the code. Introduction of a single change will require the programmer to make changes in many places in the code, which will most likely be scattered all over hundreds of lines of the code. To avoid this, developers are trying to give their code a more consistent structure.

One of the basic principles of programming is described with the acronym DRY (Don’tRepeatYourself). According to this principle, you should eliminate any repetitions, i.e. identical lines of the code. In this way, you significantly reduce the code’s susceptibility to errors and facilitate its further development.

Two most popular code refactoring methods that implement the DRY principle are as follows:

1. Encapsulating repetitive code blocks in functions
2. Encapsulating repetitive code blocks in classes

We will discuss them individually.

Grouping the code into functions

The simplest method of code refactoring, which facilitates the development and maintenance of the project, is to encapsulate repetitive fragments of code in functions. A function is a code block.

If you notice a repeating block of code in your program, you can extract it, name it, and then replace each occurrence of this block with a single-line call consisting of the block name and brackets. For this purpose, functions have been created.

You have already got to know two functions: print and type. These are functions built into Python. You have learned how to use them. Now you will learn how to create your own functions.

Assume that your program contains a block repeated in several places. Make this block print a message on the screen consisting of three identical lines of code.

```
a = 24

if a/2 == 0:
    b = a / 2
    print("First line of message") ①
    print("Second line of message")
    print("Third line of message")
else:
    b = a / 3

# Somewhere further in the code

if b != 1:
    print("First line of message") ②
    print("Second line of message")
    print("Third line of message")
```

If you would like to introduce a change in the first line of your message, you would need to do it wherever it occurs (in our case, in lines <1> and <2>). This doubles the likelihood of making a mistake when introducing changes. In addition, it is easy to skip a repetition if there are many of them. According to the DRY (Don’tRepeatYourself) principle, you should eliminate any repetition you notice in your code.

In the first step, extract the repeated lines of code and give the extracted block the

```
def print_communique():
    print("First line of message")
    print("Second line of message")
    print("Third line of message")
```

The above code defines the print_communique function. The function definition consists of the word def followed by the function name (in our case, it is print_communique), brackets and a colon. The indented code block below is called the function body. The function body is a code that will be executed every time when the function is called. The function definition is placed at the beginning of the script.

In the second step, replace all instances of the repeated three lines of the code by calling the print_communique function. Call the function by entering its name followed by opening and closing brackets.

```
print_communique()
```

Sometimes you can insert input parameters between the brackets. The input parameters of the function will be discussed later in this chapter. After refactoring, the program will have the following form:

```
def print_communique():
    print("First line of message") ①
    print("Second line of message")
    print("Third line of message")

a = 24

if a/2 == 0:
    b = a / 2
    print_communique()
else:
    b = a / 3

# Somewhere further in the code

if b != 1:
    print_communique()
```

① After refactoring, making a change in the first line of the message will require you to introduce the change in one place in the code only.

A function can be compared to a black box into which you put something. The black box processes what you have put in it and returns the result.

Our first function, i.e. print_communique, did not take any input parameters and did not return anything. It only printed a three-line message on the screen.

Our second function, i.e. double_input, will take an input parameter and return the result. Its task is to return a doubled value of the input parameter.

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

The double_input function accepts one input parameter. Inside the function body, you will refer to this parameter using identifier x. The function has a single-line body. After the return statement, insert what you want your function to return. In our case, it is a double value of the input parameter.

The value returned by the function can be printed on the screen or transferred to a variable. Call your function for several arguments:

```
print(double_input(3))
a = double_input(10)
b = double_input(25)
c = double_input(2.0)
print(b)
print(c)
```

After the above code is implemented, the following result will be displayed:

```
> 6
> 50
> 4.0
```

Let us analyse one more function. Our next function, i.e. sum_up, takes two parameters at the input and returns their sum at the output. The program that implements (executes) this function has the following form:

```
def sum_up(x, y): ①
    return x + y ②

a = sum_up(3, 4) ③
b = sum_up(2.5, 7.5) ③
c = sum_up(4, -2) ③

print(a) ④
print(b) ④
print(c) ④
```

① Define a function called sum_up which takes two input parameters: x and y.

② Create the body of the sum_up function. In this line, both parameters are added to each other, and the result is returned to the outside of the function using the return statement.

③ Call the function three times for different values of input parameters. Assign the returned output values to variables.

④ Print the values stored in variables a, b, c on the screen.

As a result of the script execution, three values will be displayed on the screen:

```
7
10.0
2
```

Grouping a code into classes

Closing repetitive code blocks into classes is another common method for code refactoring. The idea of structuring the code as a class is a natural extension of the trend (discussed in the Foreword) to give a programming language (and thus the code written in that language) an increasingly human-friendly structure. Classes are an attempt to make our perception of the surrounding world reflected in the code.

When you watch “Lassie Come Home” movie, you know that the long-haired collie that appears on the screen is a dog. You did not need anyone to explain to you that this particular dog you saw for the first time (unless you watch this film again) was a dog. I know this, although we have never seen a model dog. Such a model dog does not even exist. Dogs can be very different. Poodles look completely different from Great Danes, and these are very different from Chinese Crested Dogs. Moreover, if such a dog loses all its four legs and the tail in an accident, it will not cease to be a dog. How can we indicate without anyone’s explanation that the specific object we can see for the first time in our life is a dog? What is the definition of being a dog? And what about a table? What is a table? Every day you encounter many implementations of the Table class. A canteen table differs from the rehab table, and the latter differs from the table attached to the back of an airplane seat. And yet, all these objects are called tables and you can easily recognise them. What is a Table class? Is a table something with four legs and a top? Not necessarily. A table can have two legs, one leg, or be suspended. And if you put a tablecloth on a thick tree trunk and add a chair next to it, will not this trunk become a table?

The above reasoning leads us to conclusion that we identify the objects surrounding us not only on the basis of their characteristics (referred to as attributes), but also on the basis of the functions (methods) they perform. Sometimes you can recognise a dog only by the fact that it is barking, wagging its tail and is being led on a leash. The same applies to classes in programming. A class is a general entity, a pattern that defines a concept. For us it will be a Dog or a Table. Objects are various implementations of this class. The table at which you had breakfast today is just one of the implementations (objects) of the Table class, while Buddy, your neighbours' dog that you saw when you were leaving your house yesterday morning, is an object belonging to the Dog class.

Let us program Buddy the dog. Buddy will be our Dog class object. First, let us define the empty Dog class.

```
Dog class :
   pass
```

To create an object of your class, call its class (just as you call functions).

```
dog1 = Dog()
```

Creating an object is called its initialisation. Immediately after initialisation, the Dog class object is assigned to the variable dog1. But for now our object cannot do anything. Its template is empty. Let us think about the skills that you would like your Dog class objects (Buddy, for example) to have. Let all the dogs we create can bark. Object’s skills are called methods. A method is a special function that:

1. Is defined inside the class
2. The first parameter of the method is the word self

In the bark method, a sound file with barking could be played. However, not everyone has such a sound file on hand, so let the barking be expressed with text “Woof, woof!” printed on the screen.

```
Dog class:
    def bark(self):
        print("Woof, woof!")
```

Now the objects of your class can bark.

```
dog1 = Dog()
dog1.bark() ①
```

① To call the object’s method, put a dot after the object itself and then write the name of the method you want to call, plus the brackets.

In addition to skills, objects can also have properties (attributes). What properties would you like your Dog class objects to have? Maybe a name? All attributes are placed in the special *init* method, and a specific notation is used.

```
class Dog():
    def __init__(self, name): ①
        self.name = name ②
    def bark(self):
        print("Woof, woof")
```

① Place in brackets all the attributes that you want to assign to your objects (here it is only one, i.e. a name, because self is the default attribute that we place in each method).

② Assign these attributes to another variable by prefixing it with the word self and a dot. From this point, you can use the object’s notation to refer to the name attribute of your class objects. Let us check how it works.

```
dog1 = Dog("Buddy") ) ①
dog1.name
dog1.bark()
```

Object attributes are entered inside brackets during initialisation (in the same way as we have placed input parameters to the function). Now the objects of your class can bark and they have a name. Things that objects have (e.g. a name) are called their attributes. Things that objects can do (e.g. barking) are called their methods. Attributes and methods are called object fields.

In this concept, methods are often thought of as functions that change the value of a given attribute (that is, the state of the object). For example, let us teach the Dog class objects to eat.

What can the eat method do? It can change your dog's state from hungry to fed. Let us introduce a new attribute to our class, which states whether the dog is hungry: is_hungry. Next, the eat method will check whether the is_hungry attribute of the Dog class object is set to True. If so, it will change this attribute to False. In this way, your dog will no longer be hungry, i.e. from the hungry state (is_hungry = True) will go into the fed state (is_hungry = False).

```
class Dog():
    def __init__(self, name, is_hungry):
        self.name = name
        self.is_hungry  = is_hungry
    def eat(self):
        if self.is_hungry  == True:
            self.is_hungry  = False
```

And let us see if it works.

```
dog1 = Dog("Buddy", True)  ①
dog1.is_hungry   ②
dog1.eat() ③
dog1.is_hungry  ④
```

Here is what have done:

① We created an instance (object) of the Dog class with the attributes name = "Buddy" and is_hungry = True, and then assigned the created object to variable dog1

② We asked about the value of attribute is_happy of object dog1 (to check whether dog1 is hungry). True was displayed.

③ We called the eat method for our object.

④ We asked again about the value of attribute is_hungry for object dog1. This time False was displayed. Our eat method works as intended. This is how we have taught our dogs to eat. In this way we can teach our dogs everything. You can teach them to fly or to eat with chopsticks. You just need a little more code and a lot of imagination.

For the sake of clarity of the lesson's material, you can use names in your native language in the code. However, it is a good programming practice not to mix languages and to write code only in English. Now when the introduction is behind us, let us try to apply good practices only.

Exercises

1. Create a Person class
    - add fields: name, surname, gender, age, personal ID number
    - add a method to check whether a person has reached the retirement age (for women take retirement age >= 60, for men >= 65)
    - add a method that returns the age difference between a given person and another person:

- make the method accept a Person type parameter
- It should not return negative values as the difference of years.
- Add a method that calculates and returns the number of years remaining to the retirement.

Conclusion

This Workbook presents the basics of programming. It is a great start of the programmer's road.
The road that is interesting and endless. There are many forks on it, surprising turns and fascinating recesses. In the Internet and at bookshops you can find many excellent guides and courses discussing its individual sections. There is nothing to wait for. Just hit the road.

After reading this Workbook, you know how to give instructions to a computer. You are able to use the tool that performs billions of operations per second with unprecedented precision. It gives you a tremendous power and a big responsibility. What will you do with it? Write a cool program!

Appendix A

Running the program you have written requires additional software. In many studies, the reader is required to install this software before writing the first program. The installation and configuration process can be tedious. In the event of failure, it often discourages and demotivates the reader even before writing the first line of a code. Today it is no longer a necessary stage. Fully configured, ready-to-use development environments are available in the Internet. For the needs of this Workbook, one of them will suffice. However, for regular code writing, it is necessary to install and configure the development environment on your computer.

Any text editor (e.g. Notepad) is good to write a Python program. To run the written program, you need a Python interpreter, i.e. a program that translates a code written in Python into a string of zeros and ones. The Python interpreter can be downloaded from the official Python website. In addition, Integrated Development Environment (IDE) is often installed instead of the regular text editor.

AUTHOR'S COMMENTARY: Students get the software installation instructions at the beginning of the course. I have not read it, but I have not met a student yet who would complain about it, so it is probably enough to put it here.

The instruction covers both interactive and script modes. It ends with running the “Hello, World!” program in Pycharm.

Appendix B

Summary of types

Table 1. Types of data

Designation	Name	Examples of valid literals
<class 'int'>	Integer type	10, -768, 080, -0490, 0x260
<class 'float'>	Floating point type	0.0, 15.2, -21.9, 32.2e18, 70.34E-5
<class 'complex'>	Complex type	1j, 0, 2.3+4.5j, -34j, 3e-3-3j,
<class 'bool'>	Boolean type	True, False
<class 'None'>	None	set(), set(2), set()
<class 'bytes'>	Bytes	bytes(), bytes(4), bytes([1,43,5]), bytes("abc", "utf-8")
<class 'str'>	String type	"", '', "Hello, world!", '123', "!@#$%", "abc"
<class 'list'>	List	[], [2,], ['abcd', 786 , 2.23, "John", True]
<class 'dict'>	Dictionary	{}, {'imie': 'john', 'code': 6734, 'wiek': 32}, {'john': '766-234-245', 'ann': '677-345-268'}
<class 'tuple'>	Tuple	(), (2,), ('abcd', 786 , 2.23, "John", True)
<class 'set'>	Set	set(), set(['abcd', 786 , 2.23, "John", True])
<class 'frozenset'>	Frozen set	frozenset(), frozenset(['abcd', 786 , 2.23, "John", True])

Summary of operators

Table 2. Arithmetic operators

Symbol	Name
+	Plus
-	Minus
*	Multiplication
/	Division
//	Integer division
%	Modulo

Table 3. Comparison operators

Symbol	Name
<	Less than
>	Greater than
<=	Less than or equal to

Symbol	**Name**
>=	Greater than or equal to
==	Equal to
!=	Not equal to
is	Is
is not	Is nott

Table 4. Boolean operators

Symbol	**Name**
and	Boolean conjunction
or	Boolean alternative
not	Boolean negation

Table 5. Bitwise operators

Symbol	**Name**
<<	Left shift
>>	Right shift
|	Bitwise alternative
^	Bitwise XOR
~	Bitwise negation

Summary of keywords

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

CONGRATULATIONS!

YOU WERE COMPLETED OUR WORKBOOK

software **development** academy