POLISH NATIONAL AGENCY FOR ACADEMIC EXCHANGE

Publication financed by NAWA
as part of activities complementary to those undertaken by UJD within the COLOURS alliance

Fundamentals of Python programming

Examples and practice exercises

Lidia Stępień

Marcin R. Stępień

Hubert Dróżdż

Jan Dlugosz University in Czestochowa

Fundamentals of Python programming
Examples and practice exercises

Lidia Stępień, Marcin R. Stępień, Hubert Dróżdż

Czestochowa 2024

Reviewer
dr Marcin Ziółkowski

Translator
Katarzyna Stępień

Editor-in-chief
Paulina Piasecka-Florczyk

Proofreading
Bożena Woźna-Szcześniak

Technical editing and cover design
Bożena Woźna-Szcześniak

ISBN 978-83-67984-15-7

The Publishing House of Jan Dlugosz University in Czestochowa
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl

Contents

Preface

A computer program is a detailed set of instructions that defines the actions that a computer should perform. It is created as a result of the process of creating the program's source code in a selected programming language, which is a set of rules that determine what sequences of symbols make up a computer program and what calculations describe this program. At a later stage, the source code of the program can be processed through:

- *compilation* - the source code is translated into machine language;
- *interpretation* - the source code is continuously translated and executed by an additional program called an interpreter.

Then we will say that the programming language subject to compilation is a compiled programming language, and that subject to interpretation is an interpreted programming language.

Python is an interpreted high-level programming language created by Dutch programmer Guido van Rossum in 1990. The language was named after the BBC television program "Monty Python's Flying Circus". Python is developed as an Open Source project, and its interpreters are available for various operating systems. Python is one of the youngest, but also most commonly used programming languages today. It has a fairly easy syntax, relatively few keywords, and a very rich library base, with the help of which even very complex programming projects can be created. Python has rich possibilities for both procedural and object-oriented programming. Another advantage is that the keywords used in this language are identical to those in other modern high-level languages such as C++, JAVA or PHP. However, compared to these languages, creating programs in Python is more intuitive and does not require a beginner to have extensive computer knowledge or remember many editing details. On the other hand, this language has some interesting solutions that other languages do not have, including the existence of a computational type for complex numbers, the exponentiation operator, a default input type of `str`, a very convenient data structure `list`, and an essentially unlimited range of numeric data.

Python is a multi-paradigm programming language with versatile applications, optimized for code readability, concise syntax, and software quality. Python is currently one of the most popular programming languages. It is used in various fields, from web development and data analysis to artificial intelligence and machine learning. The large community and wealth of libraries make Python ideal for both beginners and advanced programmers.

Based on many years of teaching experience in teaching various programming languages to first-cycle computer science students, and taking into account the varying levels of programming advancement of students starting their studies, it can be stated with certainty that learning Python as the first programming language seems to be the best solution. Among other things, thanks to Python, students learn to take care of code readability and it is easier for them to transfer these skills to other programming languages that no longer have such restrictive requirements.

The proposed manual is a practical guide focusing on the basics of programming in Python. Its primary goal is to develop programming skills in Python by discussing basic programming constructs and data structures and providing rich illustrations using appropriately selected examples. In addition, tasks for self-solving will serve to consolidate the acquired skills. The reader will be able to use the skills acquired through studying this manual in any Python-based software system encountered.

This manual could not have been created without the invaluable help of many people, but **dr hab. Andrzej Zbrzezny prof. UJD** deserves special recognition. We could always count on his support and suggestions resulting from many years of work with students, as well as on inspiring us with his ideas. We are grateful for the opportunity to draw from the resources of such rich scientific and teaching experience.

Chapter 1

Introduction to Python

In this chapter, the reader will be introduced to the basic information about the Python language - from installing Python on various operating systems, through the concepts of objects, variables, operators and expressions in Python - to working with the interpreter.

1.1. Installing Python 3 interpreter

The latest version of Python can be downloaded from the appropriate sub-page of the language's website `http://www.python.org`. The details of installing Python vary depending on the platform, and it is not our intention to go into detail here, instead, we refer the interested reader to the above-mentioned website. In this manual, we assume that the reader already has Python (in version 3) installed on their computer, and for those who do not yet have it, we will briefly describe how to install it on the two most popular operating systems - Windows and Linux.

On Linux systems (including Ubuntu), Python 3 is a standard, built-in component of these platforms. On Arch Linux, Python 3 can be installed in the terminal using the command: `# python -Sy python3`.

For Windows, the Python 3 interpreter can be downloaded for free: `https://www.python.org/downloads/windows/`. The process then is as simple as running the appropriate file (depending on the operating system version and its architecture) and answering all the prompted questions `Yes` (`Yes`) or `Next` (`Next`). It is important that when running the installation file, you indicate the need to create an environment variable with the path to the interpreter so that you do not have to do it yourself later.

1.2. Python language - the basics

Python programs are written in text files and saved with the `py` extension, so you can work in a regular text editor, such as Notepad. The most convenient way, however, is to install an integrated development environment (IDE, e.g. Jupyter, PyCharm, Wings, Visual Studio Code, VIM, or (in Windows) the Python IDLE environment supplied with the interpreter during installation), which contains not only an editor for writing programs but also an interpreter, a set of necessary libraries and a graphical interface through which you can easily interpret, correct and run programs.

Python 3.x accepts characters from the Unicode character set in the `UTF-8` system in scripts. A Python program is a sequence of instructions. This means that not only the instructions in the program are important, but also their order. Instructions (language keywords) are written in lowercase letters. A feature that distinguishes Python from other languages is the use of indentation to separate blocks of source code, which increases its readability and clarity. A block must contain at least one instruction. In the case when we don't yet know what code should be in the block, we can use an empty `pass` instruction (or ellipsis instruction `...`). Python detects the beginning and end of a block based on the indentation of its instructions. Therefore, statements indented to the same depth as a space or tab are treated as a block, although according to the `PEP8`-Style Guide for Python Code `https://peps.python.org/pep-0008/`, using 4 spaces instead of a tab is suggested.

In Python, it is values, not variables, that have types, which means Python is a dynamically typed language. Every data is represented by an `object` or by a relationship between objects. Every object has an identity, a type, and a value. Once an object is created, its identity never changes. You can think of an object's identity as the object's address in memory. It is possible to compare the identities of two objects using the `is` operator. The built-in function `id` returns an integer value representing the object's identity - in the standard implementation, this function returns the object's address, converted to a numeric value. The type specifies the set of attributes and operations that can be performed on the object and defines the set of allowable values for the object. The type of an object (which is also an object), like its identity, cannot change. The type can be retrieved using the built-in function `type`.

In Python there are the following built-in data types:

1. `int` – integer numbers, e.g. `-19, 0, 1` etc., you can use the underscore character `_` as a separator (every three digits) in large numbers,
2. `float` – floating-point numbers (real), e.g. `3.14, -44.99` etc.,
3. `complex` – complex numbers, e.g. `1+4j, -2j, -39-34j` etc.,

4. `str` – character strings (e.g. words) - this is the default type of entered data, e.g. `"Ala", 'Ola',''` etc.,
5. `bool` – logical values - contains only two values `True`(1) and `False`(0).

Variable in Python is a name that is a reference to an object. A variable name is an identifier that starts with an underscore or a letter followed by any number of case-sensitive letters, numbers, or underscores. The following keywords are reserved and cannot be used as variable names:

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

There are various conventions for naming variables, but it is considered a good practice to use `snake_case`, where names are defined using lowercase letters and words are separated by the underscore character `_`. When we have several variables with similar meaning, we add numbers at the end, e.g. `name1, name2`. The name of the variable should express its purpose. It is correct to use English names (always in commercial solutions), and, in the case of non-English language-specific names - diacritics are not allowed.

To create a variable and assign a value to it, use the following assignment statement:

```
variable = expression
```

The variable `variable` is assigned a reference to the object created as a result of evaluating the expression `expression`. The assignment to the variable is not printed by the interpreter.

In Figure 1.1 a variable named `a` is created, which is a reference to an object in memory that stores the value 7. Then a second variable named `b` is created, which by assigning `b = a` is a reference to the same area of memory that the variable `a` points to. When a new value is assigned to the variable `a`, an object storing this value is created in memory, to which the variable `a` is bound. Only the variable `b` remains bound to the object storing the value 7.

In Python, it is possible to assign values to multiple variables at once. For example, if we wanted the variables `a, b, c, d` to have the value 12 at the same time, we could do this either with four separate assignments, or we could use a single statement:

```
a = b = c = d = 12
```

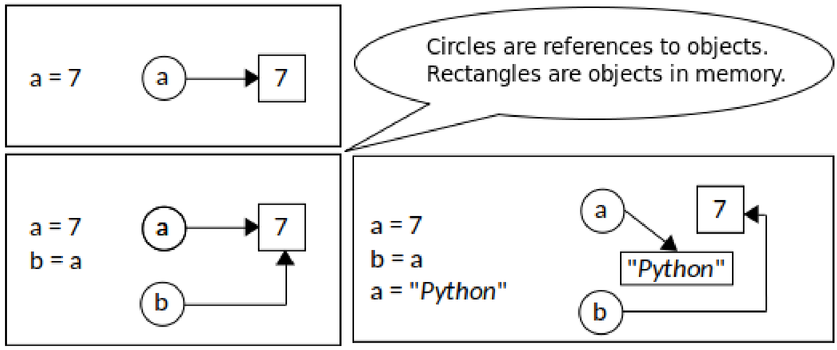

Figure 1.1: References to objects and objects in Python

If we wanted to specify different values for `a` and `b`, we could also do that on one line. In that case, we first specify the variables to which we want to assign values, separating them with commas, adding an equal sign, and specifying their values one by one, e.g. `a, b, c = 1, 2, 3` One use of this notation is to swap values so that `a` takes the value of the variable `b` and vice versa: `a, b = b, a`

In Python, expressions are built using:

- Arithmetic operators:
  - addition $+$ i subtraction $-$,
  - multiplication $*$ and division $/$,
  - integer part of division $//$ and remainders from division $\%$,
  - exponentiation $**$.
- Comparison and logic operators:
  - Comparison operators:

    $==$ - equal,

    $!=$ - not equal,

    $<$ - smaller than,

    $>$ - greater than,

    $<=$ - smaller or equal,

    $>=$ - greater or equal.
  - Logical operators in descending order of precedence:

    `not` (negation),

    `and` (conjunction - "and") as well as

    `or` (alternative - "or").

Python operators have a precedence:

- Parentheses have the highest precedence. They are used to force an expression to be evaluated in a given order.

- Exponentiation has the next highest precedence.
- Multiplication, division, integer division, and remainder have the same precedence, which is higher than addition and subtraction, which also have the same precedence.
- Operators with the same precedence are evaluated from left to right. In algebra, they are said to be left-associative.
- The exception is the exponentiation operator, which is right-associative.

Mixed operand operators convert:

- boolean operands to integer, floating-point or complex,
- integer operands to floating-point or complex,
- floating-point operands to complex.

Python also supports a special, optional form of assignment that is often useful to programmers in practice. This is the augmented assignment, which is a shortcut that combines an expression and an assignment in a concise way. For example, the assignment `i += 1` has the same effect as `i = i + 1`. Note that an augmented assignment can only be applied to a variable if it appears on both sides of the assignment. All of the arithmetic operators listed above can be used in augmented assignment expressions.

In classical logic, each statement can take only one of two logical values: `true - 1` or `false - 0`. In Python the type `bool` is a sub-type of type `int`, so calling the built-in function `issubclass`(`bool`, `int`) will return `True`:

```
>>> issubclass(bool, int)
True
>>> issubclass(int, int)
True
>>> issubclass(int, float)
False
```

In almost all contexts, the boolean values `False` and `True` behave like the values `0` and `1`, respectively. The exception is that when a boolean value is converted to a string, the string `'False'` or `'True'` is returned. In Python, any object can be treated as a boolean value `True` if necessary. The following objects are treated as the value `False`:

- `None` - a special value that represents no value, an undefined value;
- `False`;
- zero of any numeric type, for example: `0, 0.0, 0j`;
- any empty sequence, for example: `'', (), []`;
- any empty mapping, for example: `{}`;
- instances of user-defined classes, if the class defines a `__bool__` method or a `__len__` method and these methods return the boolean value `False` or the integer zero for these instances.

All other values are treated as the boolean value `True`.

The built-in function `bool` returns the boolean value of each object, for example:

```
>> bool(None)
False
>> bool(6)
True
>> bool("")
False
>> bool("Python. Hello World!")
True
>> bool([])
False
>> bool([10, 20, 30, 40])
True
```

In Python, you can chain comparisons together, as you would in mathematics, into a single expression of the form:

$$a < b < c < d$$

called a cascading comparison. This is equivalent to the expression:

$$a < b \text{ and } b < c \text{ and } c < d$$

Cascading comparisons can be of any length, but the implicit presence of the conjunction `and` means that the computation stops immediately after determining whether the truth of the entire expression can be uniquely evaluated, e.g. in the expression:

$$a < b < c < d$$

if $a < b$ is false, then the entire expression is false and the remaining comparisons are not evaluated.

An essential part of creating code in any programming language is writing comments. In Python, we distinguish the following comments:

- Single line comment beginning with the `#` symbol, eg.

```
# single line comment
# a = 1
```

- Multi-line comment - text is placed between triple single quotes `'''` or triple double quotes `"""`.

```
'''
multiline comment using single quotes
a = 1
```

```
b = a
'''

"""
multiline comment using double quotes
a = 1
b = a
"""
```

The `print` function is used to print argument values to the standard output (screen). Its arguments can be strings, literals, data structures or variable names. Additionally, it supports arguments with keywords that allow for the use of special display modes. By default, the `print` function inserts a space separator between the displayed arguments and, at the end, inserts a newline character. Therefore, calling this function without arguments - `print()` - results in inserting an empty line on the screen. You can change these settings by assigning new values to the `sep` and `end` parameters.

Listing 1.1: *Using* `print` *function*

```
print("I LOVE PROGRAMMING IN PYTHON")
print("I LOVE PROGRAMMING IN PYTHON", end='')
print("I LOVE PROGRAMMING IN PYTHON")
```

Listing 1.1 shows a program that prints three identical messages to the screen using the `print` function, where the second function additionally changes the default way of inserting a newline character after the printed message to a blank character. This causes the next message to appear on the screen, starting immediately after the last character of the preceding message.

The `input` function returns a value read from the standard input, i.e. the keyboard, by default in the form of string. The argument is a string (`string`), which will be displayed on the user's screen as a prompt, as illustrated by the code in listing 1.2.

Listing 1.2: *Using* `input` *function*

```
a = input("Enter integer number: ")
```

In order for the read value to be of different type than the default `str` it is necessary to convert (cast) it to the desired type, as shown in Listings 1.3 and 1.4, converting the read value to an integer and a real number, respectively.

Listing 1.3: *Entering integer numbers*

```
a = int(input("Enter integer number: "))
```

Listing 1.4: *Entering floating point numbers*

```
b = float(input("Enter floating point number: "))
```

Listing 1.5: *Printing out the results of calculations*

```
1 a = int(input("Enter first number: "))
2 # Enter first number: 12
3 b = int(input("Enter second number: "))
4 # Enter second number: -33
5 print("\nThe sum ob numbers:",a,"and",b,"is",a+b, sep='_')
6 # Sum_of_12_and_-33_is_-21
7 print(a,"+",b,"=", a + b)
8 # 12 + -33 = -21
```

The program in listing 1.5 asks the user to enter two integers and then prints them and their sum to the screen. It does this in two ways. On line 5, in words using the underscore character as a separator and ending with a double end-of-line character, and on line 7 according to mathematical notation. The order of arguments in the `print` function is consistent with the order of information appearing on the screen. Variables are separated from strings by a comma and indicate that the values to which they refer are to be inserted in their place. The calculated sum is also an arithmetic expression separated by a comma. Before the sum of the indicated variables is printed to the screen, their values are retrieved and the value of the expression is calculated. Since the default separator between arguments in the `print` function is a space, we do not insert additional spaces within the strings. Information that appears on the screen during program execution is placed in comments. In the first and third lines, the user entered sample integers and confirmed them with the `Enter` key.

1.3. Working with interpreter

There are two ways to work with Python. In addition to interpreting scripts, we also have the option of using interactive mode. This solution seems very useful when we want to quickly test the operation of usually not very extensive code we entered, check the operation of certain functions or commands, or use the interpreter as a regular calculator, which is frequently the situation. Using interactive mode unfortunately has a significant disadvantage. Commands entered in this way are not stored, so they cannot be executed again without re-entering them.

Starting an interactive Python session varies slightly depending on the platform you're working on. On Linux, all you need to do is type `Python3` in the shell. First, you'll see a welcome message that includes the Python version you're currently working with and some copyright information. Then, you'll see a prompt, which is usually a set of three greater-than signs (>>>). This means that Python is waiting for user interaction. When you enter a multi-line statement, Python changes the prompt to three dots (...) while it waits for the remaining lines. To exit the session, use the key sequence `Ctrl+D` (on Linux) or `Ctrl+Z` (on Windows). Here's an example of what the messages look like when you start working with the interactive mode:

```
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We will now present examples of how to use the interactive mode. First, let's try using it as a calculator.

```
>>> 2 + 3          # calculating the sum of numbers 2 and 3
5
>>> 3 * 4          # calculating the product of numbers 3 and 4
12
>>> 2 ** 3         # calculating the third power of number 2
8
>>> 5 / 2          # calculating the quotient of numbers 5 and 2
2.5
>>> 2 ** 0.5       # calculating the square root of number 2
1.4142135623730951
>>> (30 + 3*4) / 2 # the result of the operation will be a float number
21.0
>>> 2 * 3.25 - 3   # example of performing an operation on data
3.5                # of different types
>>> 1_000_000 * 2  # calculating the product of numbers 1000000 and 2
2000000
>>> 0b1110         # display the decimal value of the number
14                 # presented in the binary system
>>> 0b10           # similarly as above
2
>>> 0b1110 + 0b10  # calculate the sum of two binary numbers
16                 # the result is in the decimal system
```

Let's try to perform the next calculations, but this time using variables. We will assign the net amount of the given product to the variable `price`, and we will store the VAT rate in the variable `vat`. Using the interactive mode, we will determine the gross price.

We will also use the underscore character, which in this case will result in substituting the result of the last displayed expression.

```
>>> vat = 0.23
>>> vat          # we can check the value
0.23             # of the vat variable at any time
>>> price = 55
>>> price        # similarly for the price variable, we can
55               # display its value
>>> price * vat
12.65
>>> price + _
67.65
```

It is also possible that in an interactive session, we make a mistake, for example, by trying to display the value of a variable before assigning a value to it. In such a case, we will get an appropriate error message:

```
>>> new_price
Traceback (most recent call last):
    File ‘‘<stdin>’’, line 1, in <module>
NameError: name ‘new_price’ is not defined
```

As we have already seen, in the interactive mode, unlike when saving programs in files, we did not have to use the `print` statement to display the results of expressions, but only put the variable name itself. However, it is important to remember that we cannot always use these two conventions interchangeably. Consider the following example:

```
>>> text = ’Py\nthon’
>>> text
’Py\nthon’
>>> print(text)
Py
thon
```

As we can see, the `print` function removes quotes and processes special characters.

We have already mentioned the possibility of entering multiline commands. So let’s try to display six lines on the screen, each consisting of five "@" symbols.

```
>>> for _ in range(6):
... print(’’@’’*5)      # press Enter
...                     # here you can enter another
```

```
@@@@@                  # loop statement or
@@@@@                  # end a multi-line
@@@@@                  # statement by entering
@@@@@                  # an empty line, i.e. by pressing Enter again
@@@@@
@@@@@
```

Notice the change in the prompt to "..." starting from the second line of code. Also, remember to maintain the appropriate indentation for instructions that are to be executed in a loop. The end of a multi-line instruction is signaled by entering an empty line (pressing the Enter key again). Below is another example, this time we will display the numbers on one line:

```
>>> a = 5
>>> while a > 0:
...     print(a,end=' ')
...     a -= 1
...
5 4 3 2 1 >>>
```

Detailed information about loops can be found in the subsection 2.3.

1.4. Modules

A Python program often consists of several text files with Python code, one of which is the main `top-level` file, and the rest are zero or more auxiliary files - so-called `modules`, each of which is a namespace for its attributes. A top-level file contains program control instructions that use tools defined in module files, which in turn may use tools defined in other modules.

There are two ways to import a module, using following statements::

- `import` - which requires the module name to be specified relative to the names defined in it (e.g. `modul.name`) or
- `from` - which additionally copies one or more variables from the imported module to the scope in which it appears.

We will show how to import using the example of the mathematics library (`math`), from which we will import the `sin` function in two ways:

- using `import`:

```python
import math
print(math.sin(12))
#  -0.5365729180004349
```

- using `from`:

```python
from math import sin
print(sin(12))
# -0.5365729180004349
```

Importing a module consists of three basic steps:

1. Find the module file.
2. Compile it into bytecode (if necessary).
3. Execute the module code to create the objects it defines.

During program runtime, the three steps above are performed only when the module is `first` imported. Subsequent imports only retrieve the loaded module object from memory (`sys.modules`). If there is no module object in the `sys.modules` table, the import process starts, consisting of the three steps mentioned above.

The most important part of the import procedure is the location of the file to be imported. The Python module search path consists of a set of the following components:

1. Program root directory (automatically) - Python first looks for imported files in the root directory. The root directory, depending on how the code is run, will be the directory containing the top-level script file of this program being run, or the directory in which we are working (current working directory), in the case of interactive work.
   All import operations will therefore work automatically. There is a risk of accidentally hiding library modules, e.g. if the same name is used for the top-level file as the name of the imported module.
2. Directories `PYTHONPATH` (configurable) - in the next step, Python searches the directories specified in the `PYTHONPATH` environment variable, from left to right, if we have set it, because this variable is not predefined. In `PYTHONPATH` you provide a list of user-defined and platform-specific directory names containing Python code files. This setting is only important when importing files between different directories.
3. Standard library directories (automatic).
4. Contents of all `.pth` files (configurable) - Python allows users to add directories to the module search path by placing them one per line in a text file ending with `.pth`. This is an advanced installation option that provides an alternative to setting the `PYTHONPATH` environment variable.
   For more information, see the Python standard library documentation - in particular the `site` module, which allows you to configure the location of Python libraries and path files (the documentation describes the expected locations of path files, among other things).
5. The `site-packages` directory for external extension packages (automatic) - Python adds the `site-packages` subdirectory of its standard library to the module search

path. By convention, this is where most third-party extensions are installed, often automatically by the `distutils` tool. Because their installation directory is always part of the module search path, clients can import modules for such extensions without having to set the path separately.

After importing the standard library module `sys`, you can view the built-in `sys.path` list and see how the module search path is configured on your computer, e.g.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/usr/local/lib/python3.10/dist-packages',
'/usr/lib/python3/dist-packages']
```

This list provides for manual customization of search paths for scripts using the `sys.path.append` or `sys.path.insert` methods - with only one run of the program surviving in this modified form.

In addition to the module name, you can also specify the directory path in the import operation. The directory with Python code is called `package`, hence `package import`. Importing packages involves placing the path of names separated by dots in the `import` (`from`) instruction instead of the file name (module):

```
import dir1.dir2.module_name
from dir1.dir2.module_name import x
```

The path with dots corresponds to a filesystem path to a file named `module_name.py` (or similar).

Any directory listed in the path of a package import statement may (and must, until Python 3.3) contain a file named `__init__.py`, otherwise the import operation will fail. However, the parent directory need not contain this file, since it is not listed in the `import` statement itself, and the `__init__.py` file itself may be empty.

If the file structure looks as follows:

```
dir0/dir1/dir2/module_name.py
```

and the `import` statement is written as:

```
import dir1.dir2.module_name
```

following rules are used:

- `dir1` and `dir2` must contain a `__init__.py` file;
- the parent `dir0` directory does not have to contain a `__init__.py` file; if it does, it will be ignored;
- `dir0`, not `dir0/dir1`, must be in the module search path `sys.path`.

`__init__.py` files can contain Python code that will be run automatically the first time the directory is imported by the program and will activate the initializations required by the package.

Consider the following directory structure:

```
/home/user/Desktop/
 py3/              # directory in module search path
   __init__.py
   dir1/
     __init__.py
     dir2/
       __init__.py
       functions.py
```

The file `functions.py` contains functions defining three basic arithmetic operations:

```
# functions.py
def sum(a,b): return a + b
def product(a,b): return a * b
def difference(a,b): return a - b
```

We create script `modules.py` in catalogue:
`/home/user/Desktop/Python/programs`
containing:

```
import sys
sys.path.append('/home/user/Pulpit/py3')
import dic1.dic2.functions as m
print(m.sum(1,2))
```

The first time you import using a directory, Python automatically executes all the code in the `__init__.py` file in that directory. That means these files can be places to insert code that initializes the state required by the package, such as creating required files or opening a connection to a database. With `__init__.py` we declare that a given directory is a Python package. Without this safeguard, Python might choose a directory that has nothing to do with the program code, just because it appears earlier in the search path. The Python 3.3 namespace packages greatly reduce this role, because they achieve a similar effect algorithmically, scanning the path for more files. In the package import model, directory paths become true nested object paths after import. Once imported, for example, the expression `dict1.dict2.functions` works and returns a module object whose namespace contains all variables assigned by the `__init__.py` file of the `dict2` directory. For a detailed description of the namespace packages, the interested reader is referred to the specialist literature indicated in the bibliography or the Python documentation.

The `from` statement imports an entire module, similar to the `import` statement, but additionally copies one or more variables from the imported module to the scope in which it appears. This allows imported variables to be used directly without having to specify the module name in the expression. The `from` statement can break namespaces by overwriting variables used in the current scope. The form `from ... import *` is even more dangerous for this reason.

When creating program code that operates on modules, the `reload` function is often useful, which forces the module to be reimported in a situation where we want to obtain a newer version of the module's source code during programming or when the application requires dynamic adaptation to the user's needs. In the case of using the `reload` function together with `from`, the latter may cause problems, e.g. when the variable refers to a previous version of objects.

We encourage the reader interested in more information about how to import modules in Python to carefully study the documentation of this language.

1.5. Practice exercises

1. Remembering that the symbol `_` stores the last printed value, calculate the value of the expression: `((12**2 - 23)*15)/4`.
2. Assign the variable `x` values of different types: string, integer, real number and check the type of the variable using the function `type`.
3. Enter different values for the variable `x` using the `input` statement and check each time what type it is. What should be done to make the type of the variable compatible with the type of the entered value?
4. Read the values of two integers `a`, `b` and print to the screen their calculated:
   (a) sum,
   (b) difference,
   (c) product,
   (d) quotient,
   (e) integer part of `a` by `b`,
   (f) remainder of dividing `b` by `a`,
   (g) `b`-th power of number `a`.
5. Test what happens when we enter the expression:
   (a) `1/0`,
   (b) `print(x)`, where `x` has not been used before?
6. Check how multiplying text by an integer works. Multiply the string `'It doesn't matter how it starts, it matters how it ends ;)'` times `100`.
7. Calculate the value of the expression: $3 + \frac{2^2}{5 \cdot 4}$.

8. Calculate the value of the expression: $\sqrt{16} + 2^3$.
9. Use the ** operator to calculate the value of the expression $\sqrt{2} + \sqrt[4]{2} + 2^3$.
10. Check what the difference is between the results of dividing `5/2` and `5//2`.
11. Execute the following commands and describe the effect of each:

```
a = 10
a += 10
a = a + 10
print('a')
print(a)
print(a + 10)
```

12. Check the result of the command `123_000_000 + 123`. What is the purpose of the underscore (`_`) in the number?
13. What is the difference between the commands: `3/3*3` and `3/(3*3)`. Justify your answer.
14. Calculate -3^2 and $(-3)^2$.
15. Based on the entered gross amount, calculate the net amount (you can define the tax value as you wish, e.g. `VAT = 0.23` or `VAT = 0.08`).
16. Enter the values of the variables `weight` (in kilograms) and `height` (in meters), and then calculate your body mass index ($BMI = \frac{weight}{height^2}$) based on this data.

Chapter 2

Basic programming constructs

In this chapter, the reader will be introduced to basic programming constructs: conditional statements and iteration statements.

2.1. Conditional statement

The `if` conditional statement (called simply an `if statement`) in Python is a basic selection tool in which, depending on the value of the test expression, a specific action is performed. It is a compound statement in which we can freely nest other statements, including subsequent `if` statements. Python allows you to combine statements in a program in a sequential manner (executing them one after the other) and freely nested so that only those that meet certain conditions (selections) are executed.

2.1.1. Simple conditional statement

A simple conditional statement takes the form of a test, after which a block of statements is executed if the test returns true. We pay special attention to the colon character that appears in the conditional statement after the expression whose value is being tested, followed by a block that is a set of statements written after indentation.

Figure 2.1 shows a block diagram of a simple conditional statement, the general form of which is as follows:

```
if statement:
  T_instruction_block
```

In listing 2.1 we present the simplest form of a conditional statement with a condition that is always true. If the block of the conditional statement `if` is not complex, we can write a single statement on one line after a colon (and without indentation).

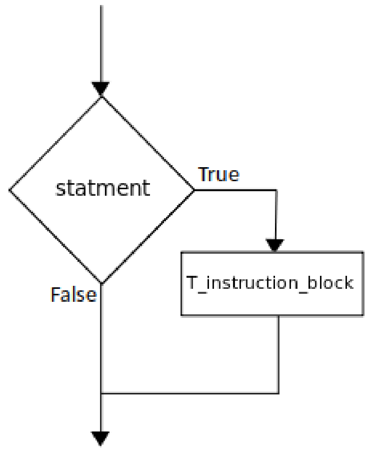

Figure 2.1: Simple conditional statement

LISTING 2.1: *condition always true*

```
if 1: print("true")
# true
```

Let's consider another example, presented in listing 2.2, in which the value of a variable supplied by the user is tested. We expect the value to be 0, which is confirmed by an appropriate message. However, if the condition being tested is false, the program will go to the instruction located directly after the conditional instruction and execute it. In a special case, this may be another `if` instruction.

LISTING 2.2: *Simple* `if` *statement*

```
1 a = int(input("Enter any integer number: "))
2 if a == 0:
3   print("Yes, you did enter ", a)
4 print("Try again some time!")
```

Using the `morse operator` you can move the instruction from line 1 of listing 2.2 to the condition of the `if` statement:

```
if a := int(input("Input any integer number: ")) == 0:
```

which will allow the use of the `a` variable in the `print` instruction.

In listing 2.3, we present the code of a program that displays the absolute value of an integer provided by the user on the screen using a simple conditional instruction and an auxiliary variable storing the value indicated by the variable b. This results from the fact that in the instruction block, when the number is negative, we change the value to its opposite value.

Listing 2.3: *Calculating absolute value*

```
b = int(input("Enter any integer number: "))
c = b
if b < 0:
  b = -b
print("|", c, "| =", b)
```

2.1.2. Conditional statement with else clause

Now consider a conditional statement with a `else` clause, which handles the case when the test of the expression put in the condition evaluates to false. Note that the `if` statement and the `else` statement are followed by colon and mandatory indentation in the statement block. Note that within the `else` clause, no expression to test is defined.

The general form of a branching conditional statement is as follows:

```
if statment:
  T_instruction_block
else:
  F_instruction_block
```

The block schema is shown on figure 2.2.

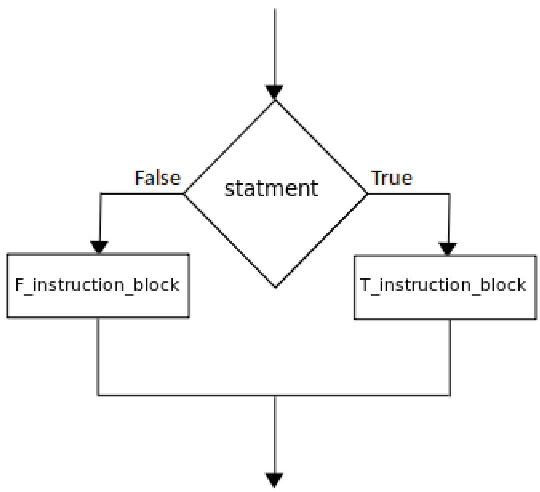

Figure 2.2: Conditional statement with `else` clause

In listing 2.4, we present a program that asks the user to enter two floating-point numbers, the first of which is to be the dividend and the second the divisor. Of course, it is necessary to check whether the divisor is not zero. If it is zero, an appropriate message is printed to the screen and the program terminates. If the divisor is not zero, and division is performed, the result is displayed on the screen before also terminating the program.

LISTING 2.4: `if-else` *statement*

```
a = float(input("Enter first number"))
b = float(input("Enter second number"))
if b == 0:
  print("Illegal zero division!")
else:
  print(a,":",b,"=",a/b)
```

One of the common tasks of logical operators is to write expressions in program code that work the same way as the conditional statement `if`. Consider the statement that, depending on the value (true or false) of the expression `X`, sets the variable `A` to `Y` or to `Z`.

```
if X:
  A = Y
else:
  A = Z
```

This is a simple construct that we sometimes want to use as a nested construct within a larger one, instead of assigning its result to a variable. Python 2.5 introduced a new statement format that allowed us to write the above construct in a simpler, more concise expression - the conditional expression, called also the `ternary operator`.

```
A = Y if X else Z
```

This expression has exactly the same effect as the previous four-line `if` statement, but it is easier to write and can be used in larger structures, providing more clarity to bigger chunks of code. You could say that it is its abbreviation. We will also show its practical use in the next chapters of this manual.

Listing 2.5 shows how to use the ternary operator to calculate and display on the screen the absolute value of an integer provided by the user.

LISTING 2.5: *The usage of ternary operator*

```
b = int(input("Enter any integer number: "))
print("|", b, "| = ", b if b >= 0 else -b)
```

2.1.3. Complete conditional statement

We will now present the syntax of the conditional statement `if`, in which all of its clauses appear (see figure 2.3, which shows a block diagram of the complete conditional statement). The `elif` clause allows us to consider another condition in the opposite condition to the one tested earlier (this is an abbreviation of the phrase `else if`). Only after all of the conditions have been tested, if each of them is false, is the statement block of the `else` clause executed. Notice that the words `if`, `elif`, and `else` are connected to each other by indentation - they are aligned vertically. It should be emphasized that the full conditional statement can only be used when the conditions following all `elif` clauses are mutually exclusive (there is no situation where at least two conditions are met at the same time).

```
if statment_1:
  instruction_block_1
elif statment_2:
  instruction_block_2
...
else:
  instruction_block_F
```

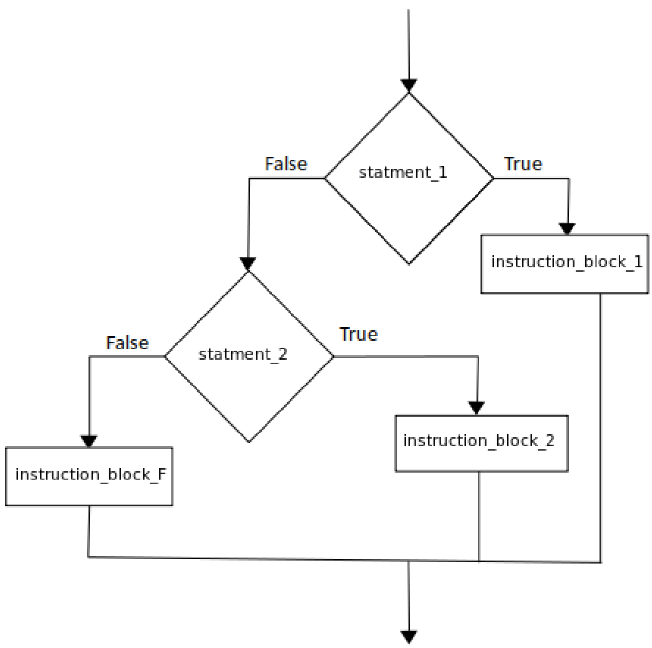

Figure 2.3: Block diagram of complete conditional statement

Consider the sample program in listing 2.6, in which the user enters the length of a side of a square, for which the user then selects whether to calculate its perimeter (requires the number 2) or its area (requires the number 1). Entering a number other than 1 or 2 prints an error message to the screen. Then, regardless of the selection, the program terminates.

LISTING 2.6: `if-elif-else` *statement*

```
a = float(input("Enter the length of the side of the square:"))
if a <= 0:
  print("Such square does not exist")
else:
  choice = int(input("1. AREA  2. PERIMETER: "))
  if choice == 1:
    print("Area: ", a*a)
  elif choice == 2:
    print("Perimeter: ", 4*a)
  else:
    print("Bad choice!")
```

2.2. Match case

Python 3.10 introduced the `match-case` selection statement. This statement takes an expression as an argument and compares its value to defined schemas. If the expression matches a defined schema, the associated statements are executed. On the other hand, if the value of the variable specified for testing is not equal to any of the defined cases, the program executes the code fragment contained in the last defined schema (often called the default), marked with an underscore `_`.

The general form of the `match-case` statement is as follows:

```
match statement:
  case schema_1:
    instruction_block_1
  case schema_2:
    instruction_block_2
  ...
  case _:
    instruction_block_if_no_previous_cases_apply
```

Consider the example of the `match-case` statement shown in listing 2.7. This statement takes an object (`rgb`) whose value the user provides, tests it against one or more matching patterns (case 'R', case 'G', case 'B'), and executes the statement block if it finds a match. If the user provides a character outside the three specified ones, the statement for the default case (`case _`) is executed and a message about the invalid selection is displayed on the screen.

Listing 2.7: `Match-case` *statement*

```
rgb = input("Choose one of the letters [R|G|B]: ")
match rgb:
  case 'R': print("RED")
  case 'G': print("GREEN")
  case 'B': print("BLUE")
  case _: print("Invalid choice!")
```

Each `case` keyword is followed by a matching pattern. Python checks for matches by going through the list of cases from top to bottom. On the first match, Python executes the block statements for that case, exits the `match-case` statements, and continues with the rest of the program.

Suppose we want to write a program (listing 2.8) that displays a message stating whether the number entered by the user is a digit or not.

Listing 2.8: *Is it a digit?*

```
is_it_digit = int(input("Enter a digit: "))
match is_it_digit:
  case 0: print('It is indeed a digit')
  case 1: print('It is indeed a digit')
  case 2: print('It is indeed a digit')
  case 3: print('It is indeed a digit')
  case 4: print('It is indeed a digit')
  case 5: print('It is indeed a digit')
  case 6: print('It is indeed a digit')
  case 7: print('It is indeed a digit')
  case 8: print('It is indeed a digit')
  case 9: print('It is indeed a digit')
  case _: print("It is not a digit")
```

In the select statement, the `|` operator, which denotes a bitwise alternative, is used to match any of a number of patterns. If the input matches any of them, the program prints out a message that the entered value is a digit. This allows us to write the program from listing 2.8 in a shortened form, as in listing 2.9.

Listing 2.9: *Merging conditions*

```
is_it_digit = int(input("Enter a digit: "))
match is_it_digit:
  case 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9:
    print('It is indeed a digit')
  case _: print("It is not a digit")
```

The `match-case` statement also allows you to define local variables that will be matched against an argument. These local variables can be referenced within the scope of a single `case` clause. Furthermore, we can specify when to match against these variables by using the `if` conditional statement.

In listing 2.10, we show how to check where a point specified by the user is located in the coordinate system.

LISTING 2.10: *Where is the point located?*

```
X = float(input("Enter x coordinate: "))
Y = float(input("Enter y coordinate: "))
punkt = (X, Y)
match punkt:
  case (x, y) if  x > 0 and y > 0:
    print('Point (', x, ',',y,') is located in Quadrant I')
  case (x, y) if  x < 0 and y > 0:
    print('Point (', x, ',',y,') is located in Quadrant II')
  case (x, y) if  x < 0 and y < 0:
    print('Point (', x, ',',y,') is located in Quadrant III')
  case (x, y) if  x > 0 and y < 0:
    print('Point (', x, ',',y,') is located in Quadrant IV')
  case (0, 0):
    print('Point is located in the center of coordinate system')
  case (x, 0):
    print('Point lies on the OX axis')
  case (_,_):
    print('Point lies on the OY axis')
```

A point is a tuple (pair) of coordinates provided by the user. If the point lies in the first quadrant (`case (x, y)`), then the matching condition must be satisfied, in which both coordinates must be positive (`if x > 0 and y > 0`, where local variables `x` and `y` are bound to elements `X` and `Y` of the tuple `point`, respectively). In the case where the point lies on one of the axes, e.g. `OX`, it is important to satisfy the matching condition, in which the second element of the tuple must be zero (`case (x,0)`). The opposite case to all the others (`case (_,_)`) is also taken into account - here, guaranteeing the position of the point on the `OY` axis.

2.3. Iteration statements

Iteration instructions, also called loops, are used in every programming language, including Python, to implement algorithms with repetitions. Such algorithms involve multiple implementations of the same operations. Each programming language has its own set of iteration instructions. In Python, we can choose from two such structures: `while` and `for`.

2.3.1. `while` loop

The `while` iteration statement is the most versatile iteration construct in Python. Virtually all programs can be written using just this statement. It causes an indented block of statements to be repeated as long as the test performed at the top level evaluates to true. If the test evaluates to false, control passes out of the block of statements covered by the `while` loop. The `while` loop block will never execute if the first test evaluates to false.

The block diagram of the `while` statement is shown in Figure 2.4. In its most complex form, the `while` loop consists of a header with a test expression, a body containing the block of statements, and an optional `else` part that is executed when control exits the loop without encountering a `break` statement to break it. The statements in the block are repeated as long as the expression tested in the header evaluates to true.

```
while test:            # the header of the loop
  instruction_block    # the body of the loop
else:                  # optional else clause
  instruction_block    # will be executed if the while loop is not
                       # stopped by the break keyword
```

The most common mistakes made by novice programmers are incorrectly set loop continuation conditions:

- the loop continuation condition is never met – the loop will never execute;
- the loop continuation condition will always be true – the loop will never end.

Such errors not only cause programs to run incorrectly, but also, especially in the case of infinite loops, put unnecessary load on computer resources (RAM) and, as a result, unstable operation of the operating system.

In listing 2.11, we present an example of an infinite loop, in which the instruction block contains the instruction `pass`. It is a placeholder without action and is used when the syntax requires a block, but we do not yet know the instructions that would create this block.

LISTING 2.11: *Example of infinite loop*

```python
while 1: pass
```

Notice that the loop body is placed on the same line as the loop header, after the colon. Similar to the conditional statement `if`, this notation only works if the body is not a compound statement.

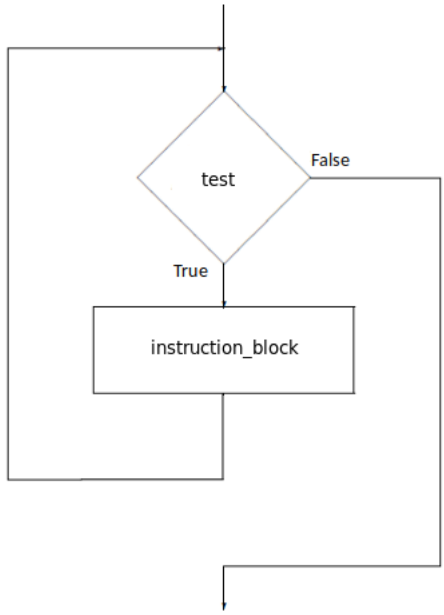

Figure 2.4: `while` loop block diagram

And another example (listing 2.12), this time of a loop that will never execute (false condition).

Listing 2.12: *Example of loop that will never execute*

```
while False: print('I won't execute anyway')
```

Python 3.x also allows you to enter ellipses (`Ellipsis` is the sole instance of the `types.EllipsisType` type) in code, in the form of three consecutive dots (`...`), in all places where an expression can appear. The ellipse, which does nothing by itself, is an alternative to the `pass` instruction or to the value `None`, e.g. we can use ellipses to initialize variable names if their specific type is not required (`a = ...`).

In the case of the `while` loop, we can specify in advance the number of times it is repeated, e.g. displaying the string `PYTHON` five times (listing 2.13 and 2.14), or the number of repetitions may depend on meeting a certain condition, e.g. we read numbers from the keyboard until the user enters a negative number (listing 2.15).

Listing 2.13: *Displaying the text* `'PYTHON'` *on the screen five times with the increamentation of the variable* `i`

```
i = 1               # counter of the num of times the text was shown
while i <= 5:       # loop will stop when i > 5
  print('PYTHON')
  i += 1            # we increase the counter by 1
```

Listing 2.14: *Displaying the text* 'PYTHON' *five times on the screen with decrementing the variable i*

```
i = 5              # counter of the num of times the text was shown
while i > 0:       # loop will stop when i == 0
  print('PYTHON')
  i -= 1           # we decrease the counter by 1
```

In the case of the programs from listings 2.13 and 2.14, they will print the text `'Python'` five times to the screen. The difference lies in the method of counting; in the first program, in the `while` loop, the variable being the counter increases its value by 1, while in the `while` loop of the second program, this variable decreases its value by 1. Therefore, the initial values for this variable are also different, as is the construction of the expressions defined in the loop header.

Listing 2.15 shows the code in which the test expression of the `while` loop is the `input` instruction, which allows the user to enter integers, and entering 0 will terminate the loop. We placed the `input` instruction in the header condition of the `while` loop and assigned the value entered by the user to the `number` variable using the morse operator. In this way, we use the data entered by the user as the values of the expression being tested.

Listing 2.15: *Expects positive integer number*

```
while(_number:=int(input("Enter positive integer number: ")))<=0:
    print("The number is not positive. Try again!")
print("You have entered positive integer number:", _number)
```

Another example of using the `while` loop is presented in listing 2.16, where there is a program code in which the user enters any natural number. If the entered value is not positive, the program prints an appropriate message and terminates. Otherwise, the sum of the `n` initial natural numbers will be calculated. For this purpose, two variables will be introduced, `i` and `sum` (the initial value is always the neutral element of the operation, in this case, the value 0 for addition), which will store the number of iterations and partial sums, respectively.

Listing 2.16: *Calculating the value of the sum of n initial natural numbers*

```
n = int(input("Enter a natural number n: "))

if n > 0:
  i = 1
  sum = 0
  while i <= n:
    sum += i
    i += 1
```

```
  if n > 1:
    print(("1 + ... +" if n>2 else ’1 +’),n, "=", sum)
  else:
    print("The sum is:", sum)
else:
  print("The given number is not a natural number!")
```

In turn, during the execution of the program from listing 2.17, the user will be asked to enter any natural number. Then, if the entered value is not positive, the program prints an appropriate message and terminates. Otherwise, the product of only such `n` initial natural numbers that are divisible by 3 (the remainder of division by 3 is 0) will be calculated. The code introduces two variables, `i` and `product` (with an initial value of 1), which will store the number of iterations and partial products, respectively.

Listing 2.17: *Calculating the product of numbers divisible by 3 not greater than n*

```
n = int(input("Enter natural number n: "))
if n > 0:
  i = 1
  _product = 1
  while i <= n:
    if i % 3 == 0:
      _product *= i
    i += 1
  print("The product is:", _product)
else:
  print("The given number is not a natural number!")
```

Iteration statements often use two instructions to control the execution of an iteration: `continue` and `break`.

The `continue` instruction causes an immediate loop to proceed upstream, skipping all statements that follow it. As an example, consider the program in Listing 2.18, which prints even numbers less than a natural number `n` specified by the user. We construct the `while` loop so that it skips all odd numbers using the `continue` instruction.

Listing 2.18: *Displaying even numbers no greater than* `n`

```
n = int(input("Enter natural number n: "))
if n > 0:
  i = 0
  while i <= n:
    i = i + 1
    if i % 2 != 0:
      continue
```

```
    print(i, end = ', ' if i < n else '')
else:
  print("The given number is not a natural number!")
```

We also note the location in the code of the instruction that increases the value of the variable `i` by 1. Since the increase must occur at each loop step, it cannot be placed below the conditional instruction `if`, because the first time it is executed for the value of `i` equal to 1, the loop would execute infinitely many times.

In turn, the instruction `break` causes an immediate exit from the loop. Since the code below this instruction will never be executed, the instruction `break` can be used to avoid nesting. Consider, for example, the program from listing 2.19, in which the user enters names to display them on the screen. After entering the word `'stop'`, the loop `while`exits.

Listing 2.19: *Displays loaded words until user enters* `'stop'`

```
while True:
  word = input('Enter name <"stop" stops input>: ')
  if word == 'stop': break
```

The `else` clause of a `while` loop is executed when the `break` statement is not used in its body, even if the loop body is never executed. This allows us to eliminate the use of additional options to check whether the loop has ended. In listing 2.20, we present a program that, for an integer entered by the user, if it is greater than 1, checks whether it is a prime number and prints appropriate messages to the screen.

Listing 2.20: *Checks if a given integer is prime*

```
n = int(input("Enter an integer greater than 1: "))
if n > 1:
 i = n // 2
 while i > 1:
  if n % i == 0:
   print(n,'is not prime -',i,'divides it without remainder')
   break
  i -= 1
 else:
  print(n,'is prime')
else:
 print("The number given is not greater than 1!")
```

2.3.2. `for` loop

The `for` loop in Python is a universal iterator (more information about iterators can be found in the following sections of this manual, including the 8.4 subsection), which allows it to iterate through subsequent elements in the order in which they are placed in a given sequence or other iterable object. The `for` instruction works on strings, tuples (e.g. (1,'Ala',2)), lists (e.g. [1,2,3,4]), ranges (e.g. range(1,10), range(1,10,2)) or dictionaries (e.g. {'age': 12, 'sex':'F'}), other built-in objects that can be iterated over, and new user-defined objects that can be created using classes (see the 7 section).

The general format of the `for` iteration instruction is:

```
for goal in object:        # header, assigning an object to a target
  instruction_block        # the body of loop, using the target
else:                      # optional else clause
  instruction_block_else   # executed if break was not used
```

The `for` loop starts with a header line that specifies the target(s) of the assignment, including the object we want to traverse. After the header is the block of statements that we want to iterate over. The name used as the target in the loop header is usually a (new) variable in the scope in which the `f#or` statement is created. It can be changed in the loop body, but it will automatically be set to the next element of the sequence when the control returns to the top of the loop. After the loop, the last element of the sequence is usually assigned to this variable unless a `break` statement is used in the loop body.

The optional block of statements in the `else` clause, which works exactly the same as in the `while` loop, is executed only if the loop was not exited by a `break` statement. This means that all elements of the sequence have been processed.

The `continue` statement discussed earlier for the `while` loop also works in a similar way in the `for` loop. So the full format of the `for` statement can be represented as follows:

```
for cel in obiekt:        # assigning object elements to the target
  blok_istrukcji          # the body of loop, using the target
  if test: break          # exiting loop, skipping else
  if test: continue       # return to header
else:
  instruction_block_else  # executed if break was not used
```

The `for` loop is a counter-based loop that is easier to write and faster than the `while` loop. The built-in `range` function can be used to generate indices for the `for` loop. In Python 3.x, `range` is an iterator that generates elements on demand, so to display all the results at once, we need to wrap it in the `list` function. (For more information on iterators, see the 8.4 subsection.)

The `range` function is a function that creates a finite arithmetic sequence starting from the value `start` and ending with the value one less than `stop` (the numerical range `<start, stop)`). It can be called with one, two, or three arguments:

```
range(start, stop, step)
range(start, stop)          # step = 1
range(stop)                 # start = 0, step = 1
```

The optional third argument of the `range` function is `step`, which when used is added to each successive integer in the result. You can define the general form of any element of a range as follows:

- In the case where `step > 0`, the j-th element of the range `r` is given by:

  `r[j] = start + step * j, where j >= 0 and r[j] < stop.`

- In the case where `step < 0`, the j-th element of the range `r` is given by:

  `r[j] = start + step * j, where j >= 0 and r[j] > stop.`

Examples of ranges shown in Python interactive mode:

- range `<0,10)`;

  ```
  >>> list(range(10))
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  ```

- range `<-5,6)`;

  ```
  >>> list(range(-5, 6))
  [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
  ```

- every fifth integer from the range `<0,30)` starting from 0;

  ```
  >>> list(range(0, 30, 5))
  [0, 5, 10, 15, 20, 25]
  ```

- all integers from the range `(-10,0>` starting from 0 - the right end of the range and decreasing each subsequent number of the result by 1;

  ```
  >>> list(range(0, -10, -1))
  [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
  ```

- function `range` called with one, zero or negative argument makes sure that there are no numbers in the given range;

  ```
  >>> list(range(0))
  []
  >>> list(range(-14))
  []
  ```

- also in the range with reversed ends, there are no numbers.

```
>>> list(range(1, 0))
[]
```

Now, let's consider some examples of using the `range` function in a `for` loop. Listing 2.21 shows the code of a program that calculates and displays the sum of `n` initial natural numbers. `n` is a value supplied by the user, and if a negative value or zero is supplied, the program terminates, and an appropriate message is displayed on the screen.

LISTING 2.21: *The sum of n initial natural numbers*

```
n = int(input("Enter natural number: "))
if n <= 0:
  print('Invalid input')
else:
  s = 0
  for j in range(1, n + 1):
    s += j
  print("The sum of numbers <1,", n, ") =", s)
```

Listing 2.22 presents a program containing nested `for` loops, whose task is to draw a square on the screen made of the asterisk `'*'` with dimensions `n`×`n`, where `n` is a positive integer supplied by the user.

LISTING 2.22: *The sum of n initial natural numbers*

```
n = int(input("Enter the size of the square: "))
if n > 0:
 for i in range(n):
   for j in range(n):
     print("* ",end="")
   print()
else:
 print('Invalid input')
```

The `range` function is also used in the `random` module to generate random values from a given range - this is the `randrange` function. In listing 2.23, we show several examples of randomly selecting a single value from different ranges ten times.

LISTING 2.23: *Random number generator*

```
import random
for i in range(20):
  x = random.randrange(20)
  print(x, ' ', end='')
```

```
print()
for i in range(20):
  x = random.randrange(1,40)
  print(x, ' ', end='')
print()
for i in range(20, 0 , -1):
  x = random.randrange(1,40,3)
  print(x, ' ', end='')
print()
for i in range(20):
  x = random.randint(1,5)
  print(x, ' ', end='')
```

In the last example, the `randint` function was used, which, unlike the `randrange` function, generates random values taking into account the right end of the range.

2.4. Practice exercises

2.4.1. Conditional statement

1. Write a program that asks the user for the coefficients a and b of a linear function, and then calculates its zero. Make sure to handle all cases correctly.
2. Write a program that asks the user for an integer, checks, and prints to the screen whether it is even or odd.
3. Write a program that asks the user to enter a grade in percent and prints the corresponding standard grade to the screen in words.

```
very good          90% - 100%
good plus          80% - 89%
good               70% - 79%
satisfactory plus  60% - 69%
satisfactory       50% - 59%
insufficient       below 50%
```

4. Write a program that, for three integers `a, b, c` given by the user, finds and prints to the screen the largest of them. Try to solve the problem using the fewest possible conditional statements.
5. Write a program that, for floating-point numbers `a`, `b` and `c` that are coefficients of the equation $ax^2 + bx + c = 0$, determines and prints to the screen the real solutions of this equation. Extend the program to the case where the solutions are in complex numbers.

6. Write a program in which the user reads the radius of a circle into the variable `r`, and then the program calculates and displays on the screen the area and circumference of a circle with radius r.
7. Write a program in which the user reads the lengths of the bases of a trapezoid and its height, and then the program calculates and displays on the screen the area of a trapezoid with the given bases and height.
8. Write a program that, using Heron's formula, calculates and displays on the screen the area of a triangle, for the lengths of the sides provided by the user. Can a triangle always be built from any three segments?
9. Write a program that displays on the screen in which quadrant of the coordinate system a point given by the user lies. NOTE: include cases where the point lies in the center of the system or on the axes by displaying appropriate messages on the screen.
10. Write a program that checks for three numbers `a`, `b` and `c` entered from the keyboard whether they are Pythagorean triples. Additionally, assume that `a > 0`, `b > 0` and `c > 0`.
11. Write a program that illustrates the operation of the logical operator `or` in the format:

```
a = True
b = True
print("Examples of using the 'or' operator:")
print("True or True -> ", ..., ".", sep = "")
print("False or True -> ", ..., ".", sep = "")
print("True or False -> ", ..., ".", sep = "")
print("False or False -> ", ..., ".", sep = "")
```

    where in place of `...` insert the appropriate Boolean expression built from variables `a`, `b` and the appropriate Boolean operators.
12. Similarly to task 11, write a program that illustrates the operation of the logical operator `and`.
13. Using the `or`, `and`, and `not` operators, write a program that checks De Morgan's first law for `p` and `q` given by the user: the negation of the conjunction of two statements is equivalent to the disjunction of their negations.
14. Write a program that solves a system of two linear equations with two unknowns using the method of determinants. The coefficients are to be read from the standard input. Include all possible solutions in the program.
15. Write a program that simulates dropping a ball from a tower (without using a loop). First, ask the user for the height of the tower in meters. Assume that gravity is normal ($g = 9.8\frac{m}{s^2}$) and that the ball has no initial velocity (the ball does not move

to start). Have the program display the ball's height above the ground after 0, 1, 2, 3, 4, and 5 seconds using the formula:

$$distance\_fallen = \frac{g * x\_seconds^2}{2}$$

NOTE: The ball should not go underground.
Sample output:

```
Enter the height of the tower in meters: 100
At 0 seconds, the ball is at: 100 meters
At 1 second, the ball is at: 95.1 meters
After 2 seconds, the ball is at: 80.4 meters
After 3 seconds, the ball is at: 55.9 meters
After 4 seconds, the ball is at: 21.6 meters
After 5 seconds, the ball is on the ground.
```

16. Write a program that asks the user to enter their full name and age. The program is to provide the user with the year of birth as output. Secure the program so that neither a negative age nor a value greater than 110 can be entered.
Sample output:

```
Enter your name and surname: John Smith
Enter your age: -32
You entered incorrect data for age. Again
Enter your age: 220
You entered incorrect data for age. Again
Enter your age: 32
John Smith was born in 1992.
```

17. Write a program that takes the user's weight and height, and then calculates the body mass index ($BMI = \frac{mass[kg]}{height^2[m^2]}$).
18. Write a program that will act as a running calculator performing the following calculations:
    (a) determining the time of finishing a race on a given distance based on the loaded pace `[min/km]`;
    (b) determining the pace of running on a given distance `[min/km]` based on the assumed finishing time.

    Make sure that the user has the option to choose the distance of the race.
19. Write a program that will act as a calculator of net/gross amounts.
20. Write a program that checks whether the year entered by the user is a leap year.
21. Write a program that checks the angle between the clock hands (hour and minute) at that time, given the current time entered by the user.

22. Write a program that will verify the correctness of the check digit of the entered social security number.

2.4.2. Iteration instructions

1. For each of the following points, write a program that calculates and prints the value specified at that point:
   (a) the sum of all even numbers from `2` to `100` (inclusive);
   (b) the sum of squares of all numbers from `1` to `100` (inclusive);
   (c) the sum of powers of 2 for exponents from `1` to `63` (inclusive);
   (d) the sum of all odd numbers between `a` and `b` (inclusive), where `a` and `b` are variables that must first be loaded with two integers. For `a > b` the sum should be zero.
2. Write a program in which the user enters a natural number n, and then the program calculates and prints the sum to the screen: $1^2 + 2^2 + 3^2 + ... + n^2$.
3. Write a program that calculates the sum of all even numbers from 2 to n inclusive (n is an even number entered by the user).
4. Write a program that factors the natural number entered by the user into prime factors. For example, for the number `36` (`36 = 2 * 2 * 3 * 3`), the algorithm should print the sequence of numbers: `2, 2, 3, 3`.
5. Write a program that checks whether a number entered by the user is perfect (a perfect number is a natural number that is equal to the sum of all its proper divisors, less than that number, more information, e.g. `https://en.wikipedia.org/wiki/Perfect_number`.
6. Write a program that calculates the double factorial (`n!!`) for a natural number entered by the user `n`. The double factorial is defined as follows:

```
0!!=1
1!!=1
2!!=1*2=2
3!!=1*3=3
4!!=1*2*4=8
5!!=1*3*5=15
```

   Hint: There is no need to consider two cases of input (even, odd). You can multiply using the commutative property backwards, e.g.

```
4!!=4*2
7!!=7*5*3*1
```

   i.e. we start multiplying from the number `n`, we successively multiply by numbers 2 smaller until `i >= 1`.

7. Write a program that for the number `n` entered by the user calculates the `n`-th term of the Fibonacci sequence, i.e. the sequence defined recursively:

   ```
   a(1)=1
   a(n+2)=a(n+1)+a(n)
   ```

8. Write a program that displays a list of prime numbers smaller than the natural number `n` entered by the user.
9. Write a program that checks whether the user entered the password correctly. The user can enter the password only 5 times. The password is: `1979`.
10. Write a program that asks the user to enter a character until the letters entered can be used to create the word `MOM`. (Important: we ignore case.)
11. Write a program that prints the following figure to the screen, consisting of any character and the number of characters in the first line entered by the user, e.g. for the character `'*'` and the number of characters in the first line equal to `4` the screen will show:

   ```
   * * * *
   * * *
   * *
   * *
   * *
   * * *
   * * * *
   ```

12. Knowing that $1233 = 12^2 + 33^2$, write a program that finds all numbers from 1000 to 9999 that satisfy such an interesting relationship. The program should also count how many such numbers there are.
13. Write a program that displays the multiplication table for numbers from 1 to 100 using a nested loop.
14. Write a program that finds the largest and smallest number from the set `n` randomly generated integers (use e.g. the `randint` function from the `random` library to draw numbers) from the range 0 to 100 and calculates the average value of all the randomly drawn numbers.
15. Write a program that draws any integer from zero to 10 (use e.g. the `randint` function from the `random` library to draw numbers), and then asks the user to guess it until the user gives the correct value. Extend the program so that it provides information on which time the guess was successful or with hints such as "*The number you entered is greater/smaller than the one drawn*".
16. Write a program that is a modification of the previous program in such a way that it limits the user's ability to guess the randomly drawn number to 3 attempts.

17. Write a program that prints the first `n` natural numbers in ascending and descending order in separate columns, as shown below:

```
0 5
1 4
2 3
3 2
4 1
5 0
```

18. Write a program that prints the coordinates of all grid points inside a circle of a given radius.
19. Write a program that takes numbers from the user until the number zero is entered. The program's task is to determine the largest and arithmetic mean of the entered numbers.
20. Write a program that selects a natural number from the range `[1,1000]` and then prints all of its natural divisors.
21. Write a program that, for a natural number `n` entered by the user, prints all numbers divisible by `2` and not divisible by `3` that are less than or equal to `n`.
22. Write a program that, for a positive natural number entered by the user, prints all natural numbers `n`-digit.
23. Write a program that, for a given positive natural number `n`, prints in descending order all even natural numbers less than or equal to `n`.
24. Write a program that displays a menu on the screen in the following form:

```
0 - exit
1 - read a number
2 - display the sum
3 - clear memory
```

If the appropriate option is selected, the program should:
(0) stop,
(1) ask the user to enter the next number,
(2) display the sum of the read numbers,
(3) delete the data concerning the calculated sum.

Chapter 3

Strings

A string (*str*) is a built-in data type in Python (representing an ordered collection of characters) used to store and represent textual information and text-based byte sequences. We have already encountered strings in earlier chapters when discussing the `print` and `input` functions, among others. In this chapter, we will focus only on the most commonly used string tools and examples of their use. The reader interested in learning about all the tools for working with strings is referred to the complete documentation in the Python standard library manual.

3.1. Basic info

Python strings are considered immutable sequences, which means that their contents are ordered from left to right, and the strings themselves cannot be modified in place.

We will start by discussing two basic functions for character encoding conversion. The first is the built-in function `chr`, which, when called with the argument `m`, returns a string representing the character whose Unicode code point is the integer `m`, e.g. `chr(80)` returns the string `'P'`. The range of values for the argument of the function `chr()` is the mutually closed interval `<0, 1_114_111>`. If the argument of the function `chr()` is outside this range, the error `ValueError` is generated. In listing 3.1, we present graphic characters of numeric ranges, including uppercase and lowercase letters of the alphabet and digit characters.

Listing 3.1: *Użycie funkcji* `chr`

```
for j in range(65,91):    #capital letters of the Latin alphabet
  print(chr(j), end = ' ')
print()

```

```
for j in range(97,123):  #lower case letters of the Latin alphabet
  print(chr(j), end = ' ')
print()

for j in range(48,58):   #numerical characters
  print(chr(j), end = ' ')
print()
```

In turn, calling the `ord` function with an argument that is a string representing a single Unicode character returns an integer representing the Unicode code point of that character, e.g. `ord('A')` returns the integer `65`. The `ord` function is the inverse function of the `chr` function, i.e.

```
ord(chr(65)) = 65
chr(ord('A')) = 'A'
```

Listing 3.2 shows the codes of the listed characters in strings. This is the opposite of the `chr()` function from listing 3.1.

LISTING 3.2: *Użycie funkcji* `ord`

```
for s in "0123456789":
  print(s, ":", ord(s), end = ' ')
print()
for s in "ĄĆĘŁŃÓŚŻŹ":
  print(s, ":", ord(s), end = ' ')
print()
for s in "ąćęłńóśżź":
  print(s, ":", ord(s), end = ' ')
print()
```

In addition to numbers, Python also manipulates strings of characters, expressed in a variety of ways. Strings can be enclosed in both apostrophe and double-quote characters. Here are some ways to represent a string using the Python interpreter.

```
>>> 'computer science'
'computer science'
>>> "computer science"
'computer science'
>>> 'O\'relly'
"O'relly"
>>> "O'relly"
"O'relly
```

The interpreter prints the result of operations on strings in the same way as they are entered. They are enclosed in single quotes or double quotes and may contain other characters preceded by a slash (\) - also called an (*escape sequence*) - so as to accurately show the contents of the string, e.g.

```
\a # alarm, short sound signal
\n # move to new line
\' # apostrophe character
\" # quotation mark character
"Movie title \"Matrix\""
```

A string is enclosed in a pair of double quotes if it contains only single quotes, otherwise it is enclosed in a pair of single quotes, e.g. `"Title of the movie 'Matrix'"` or `'Title of the movie "Matrix"'`. Strings can also be enclosed in triple single quotes or triple double quotes, as the reader is already familiar with from the earlier Python comments.

Strings can be **concatenated** using the `+` operator and **repeated** using the `*` operator. This allows us to combine strings into a single string, regardless of the characters that delimit the strings:

```
>>> print("John" + ' saw ' + "a dog")
John saw a dog
```

or replicate without the need to use iteration:

```
>>> print(3 * ":) ")
:) :) :)
```

In addition to the above-mentioned ways of writing strings in code, there are also:

- raw strings - a string of characters that is preceded by the letter `r` (lower or uppercase) just before the opening quotation mark or apostrophe. This disables escape sequences, which in effect causes Python to preserve literal slash characters as they were entered. Raw strings are used, among other things, to represent directory paths in operating systems, e.g. `r"C:\my_python\task1.py"`, or in regular expressions, e.g. the pattern `r'ab{3}'` matches a string of characters starting with the letter `'a'`, immediately followed by exactly three letters `'b'` - pattern matching is handled by the `re` module;
- byte literals, e.g.
  `b'green\x01ony'`;
- Unicode literals in Python versions 3.3+, e.g.
  `u'green\u0020red'`.

3.2. Indexes and substrings

Strings are sequences of characters that can be indexed. That is, you can refer to a single character in a string using the indexing operator (`[]`) and the ordinal number of that character in the string. The first character in a string has an index (ordinal number) of 0. There is no separate type for single characters - a character is simply a string of length one. Strings in Python cannot be modified, so trying to assign a new value to an indexed position in the string generates a `TypeError` error.

```
>>> s = "programming"
>>> s[0] = "P"
Traceback (most recent call last):
File «stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Substrings can be specified using the so-called *slicing* notation of two indices separated by a colon:

```
>>> "programming"[0:3]
'pro'
>>> "programming"[3:6]
'gra'
>>> "programming"[6:]
'mming'
```

Slicing indices have useful default arguments:

- omitted first index defaults to zero,
- omitted second index defaults to the length of the string being sliced.

The slicing operation has the following useful property: substrings joined by the same ordinal, omitting the first index in the slicing range in the left join argument and the second index in the slicing range in the right join argument, `s[:i] + s[i:]` are equal to the entire string `s`, e.g.

```
>>> "world"[:3] + "world"[3:] == "world"
True
```

Incorrect slicing indices are handled quite carefully. An index that is too large is replaced by the length of the string.

```
>>> "programming"[:19]
'programming'
```

An upper bound that is less than a lower bound results in an empty string.

```
>>> "programming"[3:2]
''
```

Python allows two types of indexing - non-negative and negative numbers, and even combining both types of indexing, e.g. the string `'sandwich'` can be indexed with numbers from 0 to 7 or from -8 to -1. To determine a substring, counting from the right side of a given string, you can use negative indexes, e.g.

```
>>> "programming"[-1] # last character
'g'
>>> "programming"[-2] # second-last character
'n'
>>> "programming"[-3:] # last three characters
'ing'
>>> "programming"[:-1] # every character, except for the last one
'programmin'
```

We can also get a reversed string (written backwards) very easily. Just use the following construction:

```
>>> 'programming'[::-1]
'gnimmargorp'
```

Negative cuts that exceed the string limits are shortened,

```
>>> 'programming'[-100:]
'programming'
```

but specifying a negative single-element index that is outside the range of the string (not implying slicing) will result in an `IndexError` error, e.g.

```
>>> 'programming'[-20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

The interpreter's built-in function `len` returns the length of a string. We can use it to iterate over a string and print each character it consists of using a `while` loop (listing 3.3).

Listing 3.3: `len` *function example*

```python
word = 'Programming is fun!'
i = 0
```

```
length = len(word)
while i < length:
  print(word[i], end = '_' if i < length - 1 else '')
  i += 1
```

Due to the fact that strings in Python are iterable objects, we can use the `for` loop to print their individual characters, significantly shortening the program code (listing 3.4).

LISTING 3.4: *Using for loop to iterate over string*

```
for i in 'programming is cool!':
  print(i, end = '_' if i != '!' else '')
```

Strings can be compared using the standard relational operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. The result of the comparison follows the lexicographic order defined by the Unicode code point values of the characters involved in the comparison, e.g.

```
>>> "Windows" < "linux"
True
>>> "Windows" < "Linux"
False
>>> "Mark" < "Tom"
False
```

The `in` operator is used to test whether a given string is a substring of another string. The `not in` operator is used to test whether a given string is not a substring of another string. For example:

```
>>> "gram" in "programming"
True
>>> "program" in "programming"
True
>>> "ing" in "programming"
True
>>> "grama" in "programming"
False
```

3.3. Chosen str class methods

In Python, strings are objects of the `str` class, which has a number of built-in methods defined for operating on strings (for more information, the interested reader can refer to the Python documentation `https://docs.python.org/library/stdtypes.html#string-methods`).

In this manual, we will not present all available methods of the `string` class, but we will focus only on the most popular ones.

- `capitalize()` – returns a copy of the string with the first character changed to uppercase, e.g. we will use this function to change the first letter of a movie title to uppercase. We will do this in two ways: by printing the changed title to the screen and by assigning the changed title to a variable so that it can be used later in the program.

```
title = "journey to become the pirate king"
print('title:', title)
print('capitalize():', title.capitalize())
changed_title =  title.capitalize()
print('changed_title:', changed_title)
```

- `count(string[,start[,end]])` – returns the number of non-overlapping occurrences of the string `string` in the range `[start:end]`. The optional arguments `start` and `end` are interpreted the same as in the slicing operation.

```
title = "journey to become the pirate king"
amount= title.count("a")
print("count(\"a\"):", amount)
amount= title.count("er")
print('count("er"):',amount)
amount= title.count("a", 0, 9)
print('count("a", 0, 9):',amount)
amount= title.count("a", 0, 10)
print('count("a", 0, 10):',amount)
amount= title.count("king")
print('count("king"):',amount)
```

- `endswith(suffix[,start[,end]])` – returns the result of checking whether a string ends with the string `suffix`. If the argument `start` is present, the check starts from this character. If the argument `end` is present, the comparison ends at this character.

```
title = "journey to become the pirate king"
print('endswith("king"):',end='')
print(title.endswith("king"))
print('endswith("pirate"):',end='')
print(title.endswith("pirate"))
print('endswith("pirate",0,24 ):',end='')
print(title.endswith("pirate",0,16 ))
```

- `expandtabs([size])` – returns a copy of the string with all tab characters replaced by spaces. If size is not specified, a tab size of 8 characters is assumed.

```
title = 'journey\tto\tbecome\tthe\tpirate\tking'
print('Tab:',title)
print('expandtabs(1):',title.expandtabs(1))
```

- `find(substring[,start[,end]])` – returns the lowest index of a `substring` such that word is included in the slice corresponding to the interval `<start:end)`. The optional arguments `start` and `end` are interpreted the same way as in the slice operation. Returns -1 if word `substring` is not found. IMPORTANT: The `find` function should only be used when you want to know the position of the string `substring` in the given string. If you just want to check if the word `substring` occurs in the given string, use the `in` operator: `substring in string`.

```
title = "journey to become the pirate king"
print('find("king"):',title.find("king"))
print('find("king",0, 19):', end='')
print(title.find("king",0, 19))
```

- `isalnum()` – returns the result of checking whether all characters in a string are alphanumeric and the string consists of at least one character.

```
print('"Asd123klmP".isalnum():',end='')
print("Asd123klmP".isalnum())
print('"Asd12!@3klmP".isalnum():',end='')
print("Asd12!@3klmP".isalnum())
```

- `isalpha()` – returns the result of checking whether all characters in a string are letters and string consists of at least one character.

```
print('"A123aaa".isalpha():', end='')
print("A123aaa".isalpha())
print('"Aaaa".isalpha():', "Aaaa".isalpha())
```

- `isdigit()` – returns the result of checking whether all characters of a string are digits.

```
print('"1234".isdigit():',"1234".isdigit())
print('"1234a".isdigit():',"1234a".isdigit())
```

- `islower()` – returns the result of checking whether all letters of a string are lowercase and the string contains at least one lowercase letter.

```
title = "journey to become the pirate king"
print('title[0].islower():',title[0].islower())
print('title.islower():',title.islower())
```

- `isspace()` – returns the result of checking whether all characters in a string are whitespace characters and the string consists of at least one character.

```
print('"  \t\t  ".isspace():'," \t\t ".isspace())
print('"".isspace():',"".isspace())
```

- `istitle()` – returns the result of checking whether the string has a title structure, i.e. each word of the string must start with a capital letter and consist only of lowercase letters or non-letter characters.

```
title = 'Journey to become the pirate king'
print('title:', title)
print('istitle():', title.istitle())
title = "Journey To Become The Pirate King"
print('title:', title)
print('istitle():', title.istitle())
```

- `isupper()` – returns the result of checking whether all letters in a string are uppercase and the string contains at least one uppercase letter.

```
title = "Journey to become the pirate king"
print('title[0].isupper():',title[0].isupper())
print('title.isupper():', title.isupper())
```

- `ljust(width)` – returns a left-justified copy of the string in a string of width `width`. Padding is obtained using space characters. If `width` is less than `len(s)` the original string is returned.

```
word = 'The sun is shining'
print('ljust():', word.ljust(len(word)+10))
```

- `lower()` – returns a copy of the string with uppercase letters converted to lowercase.

```
word = 'The sun is shining'
print('lower():',  word.lower())
```

- `lstrip([chars])` – returns a copy of a string with characters removed from the beginning of the string. If the `chars` argument is not given or is set to `None`, whitespace is removed. If this argument is given and is not set to `None`, it must be of type string. The characters in the `chars` argument are removed from the beginning of the string for which this method is called.

```
word = '\t\tThe sun is shining'
print(napis, ': ', end='')
print(word.lstrip())
print(word.lstrip("\tAa"))
print(word.lstrip("\tBCD"))
```

- `partition(sep)` - splits a string at the first occurrence of the separator `sep` and returns a 3-element tuple containing the part before the separator, the separator, and the part after the separator. If the separator is not found, returns a 3-element tuple containing the entire string as the first component of the tuple and two empty strings as the remaining components of the tuple, e.g.

```
>>> a = "The sun is shining"
>>> a.partition(' ')
('The', ' ', 'is shining')
>>> a.partition('b')
('The sun is shining', '', '')
```

- `replace(old,new[,amount])` – returns a copy of the string with all occurrences of the string `old` replaced by `new`. If the argument `amount` is given, only the specified number of occurrences will be replaced.

```
word = 'The sun is shining'
print('replace():', word.replace('a','x',2))
```

- `rfind(word[,start[,end]])` – returns the highest index of the occurrence of the string `string` such that `string` is contained in the range:
`<start,end)`.
The optional arguments `start` and `end` are interpreted the same as in the slice operation. It returns -1 if `string` is not found.

```
word = 'The sun is shining'
print('rfind():', word.rfind('a '))
print('rfind():', word.rfind('a ',0,5))
print('rfind():', word.rfind('a ',0,2))
```

- `rjust(width)` – returns a copy of the string right-justified in a string of width `width`. Padding is obtained using space characters. If width is less than `len(s)` the original word is returned.

```
word = 'The sun is shining'
print('rjust():', word.rjust(len(word)+10))
```

- `rstrip([chars])` – returns a copy of a string with characters removed from the end of the string. If the `chars` argument is not given or is `None`, whitespace characters are removed. If this argument is given and is not `None`, it must be of type string. The characters that are part of the `chars` argument are removed from the end of the string for which this method is called.

```
word = '\t\tThe sun is shining\t\t'
print(napis, ': ', end='')
print(word.rstrip())
print(word.rstrip("\tAa"))
print(word.rstrip("\tBCD"))
```

- `split(sep=None, maxsplit=-1)` - returns a list of words in a string, using the separator `sep` as a delimiter. If the argument `maxsplit` is given, the resulting list will have at most `maxsplit + 1` elements. If the argument `maxsplit` is not specified or equal to -1, there is no limit to the number of splits. If the argument `sep` is given, then subsequent occurrences of it in the string are not grouped together and are considered to separate empty strings. If the argument `sep` is not specified or equal to `None`, a different splitting algorithm is used: adjacent whitespace characters are treated as one separator, and the result of the split will not contain empty strings. Consequently, splitting an empty string or a string consisting of only whitespace characters will result in an empty list. For example:

```
>>> "".split(",")
['']
>>> "".split()
[]
>>> s = "The sun is shining, the birds are\n singing"
>>> print(s)
The sun is shining, the birds are
singing
>>> s.split()
['The','sun','is','shining,','the','birds','are','singing']
>>> s.split(",")
['The sun is shining',' the birds are\n singing']
>>> s.split("is")
['The sun',' shining, the birds are\n singing']
>>> s.split(maxsplit = 2)
['The','sun','is shining, the birds are\n singing']
```

- `startswith(prefix[,start[,end]])` – returns the result of checking if the string begins with the string `prefix`. If the argument `start` is present, the check starts with this character. If the argument `end` is present, the comparison ends with this character.

```python
word = '\t\tThe sun is shining\t\t'
print('startswith():',word.startswith("\tT"))
print('startswith():',word.startswith('\t',0,2))
print('startswith():',word.startswith("\t\tT"))
```

- `strip([chars])` – returns a copy of a string with characters removed from the beginning and end of the string. If the `chars` argument is not given or is set to `None`, whitespace is removed. If this argument is given and is not set to `None`, it must be of type string. The characters included in the `chars` argument are removed from the beginning and end of the string for which this method is called.

```
word = '\t\tThe sun is shining\t\t'
print(napis, end='|\n')
print('strip():', word.strip())
```

- `swapcase()` – returns a copy of the string with lowercase letters converted to uppercase and uppercase letters converted to lowercase.

```
word = 'The sun is shining'
print(word)
print('swapcase(): ', word.swapcase())
```

- `title()` – returns a copy of the string converted to a title structure, i.e. each word of the string is converted to start with an uppercase letter with the remaining letters converted to lowercase.

```
word = 'The sun is shining'
print(word)
print('title():', word.title())
```

- `upper()` – returns a copy of the string with all letters converted to uppercase.

```
word = 'The sun is shining'
print(word)
print('upper():', word.upper())
```

- `zfill(width)` – returns a string left-padded with zeros to the specified width. If the argument value is less than the length of the string, the original string will be returned.

```
print('zfill():', "12345".zfill(20))
```

For objects of type `string`, in addition to the previously mentioned built-in function calculating and returning the length of a string (`len`), you can also use the functions `min` and `max`, which find and return the minimum and maximum elements of the string, respectively, according to the lexicographical order, e.g.

```
word = 'To be, or not to be: that is the question'
print(max(word), 'in position', word.find(max(word)))
print(min(word), 'in position', word.find(min(word)))
```

3.4. String formatting

The `format` method performs a formatting operation on a string. The string that this method is called on may contain placeholder fields delimited by curly braces `{}`. Anything not enclosed in curly braces is treated as literal text, which is copied unchanged to the resulting string. If you want to include a curly brace character in a literal text, you must double it: `{{` and `}}`. Each placeholder field contains either the numeric index of a positional argument or the argument name as a keyword. The `format` method returns a copy of the string, with each placeholder field replaced with the string-converted value of its corresponding argument. For each placeholder field, you can specify the number of positions in which the argument value should be displayed by specifying it after a colon, e.g. `{0:3}` means three positions for the displayed value. Whereas the placeholder field `{2:6.2f}` means that the displayed value will be a number of type `float`, for which we can specify the total number of positions at which it is to be displayed, including the number of decimal places. Additionally, within the designated display field, you can specify justification: `:<` - to the left, `:>` - to the right or `:^` - centered.

```python
s1 = "My name is {0}!".format("Forrest")
print(s1)
name = "Forrest Gump"
age = 18
s2 = "I am {1} and I am {0} years old.".format(age, name)
print(s2)
n1 = 4
n2 = 5
s3 = "{0:>3} * {1:<3} = {2:^6.2f}".format(n1, n2, n1 * n2)
print(s3)
```

However, if you do not specify a numeric index for a positional argument within the curly braces then Python will fill each placeholder field with the subsequent argument values, starting from the left.

```python
s1 = "My name is {}!".format("Forrest")
print(s1)
name = "Forrest Gump"
age = 18
s2 = "I am {1} and I am {0} years old.".format(age, name)
print(s2)
n1 = 4
n2 = 5
s3 = "{:>3} * {:<3} = {:^6.2f}".format(n1, n2, n1 * n2)
print(s3)
```

You can also use formatting of the argument value in the placeholder field with conversion to hexadecimal `{:x}` or binary `{:b}`.

```python
print("The decimal {0} to hex value {0:x}".format(123456))
print("The decimal {0} to binary value {0:b}".format(123456))
```

Basic information about the `format` method can be found in the documentation: http://docs.python.org/3/library/string.html.

Python 3.6 adds a new easier way to interpolate strings (include variable values directly into strings) - `f-strings` (PEP498). To create an `f-string`, precede the string with the letter 'f'. The string itself can be formatted in much the same way as in the `format()` method. Similarly, we can use placeholder fields enclosed in curly braces in the string, e.g.

```
>>> major = 'computer science'
>>> f"Major: {major}"
'Major: computer science'
```

We can also format the displayed string by dynamically specifying the width of the display field and the precision for numbers of type `float`.

```
>>> total = 45.758
>>> width = 10
>>> precision = 4
>>> f"Your total is {total:{width}.{precision}}"
'Your total is      45.76'
```

We created three variables. The first one is some real number, and the other two are integers. Then we create `f-string`, in which we place the variable `total` with formatting: `width` of the field should be 10 characters, and `precision` of the real number is defined as 4 digits: two digits of the integer part plus two digits after the decimal point (the value is rounded). The same variable in `f-string` can be used multiple times.

3.5. Practice exercises

1. Write a program that creates and prints to the screen a word consisting of lowercase letters of the alphabet of a string provided by the user, but in the reverse order of their occurrence, e.g. for the string: `'The Sun is Shining'` the program will print: `'gninihsinueh'`.
2. Write a program that prints to the screen only uppercase letters of the alphabet found in the string provided by the user.
3. Write a program that counts and displays to the screen the number of characters other than letters of the alphabet (we do not distinguish between case) found in the string provided by the user.
4. Write a program that prints to the screen the character that most frequently appears in the string provided by the user.
5. Write a program that works with two strings provided by the user and prints to the screen a word consisting of unique characters occurring in both strings.

6. Write a program that compares two strings provided by the user and prints to the screen the message:
   `'The same strings'` – if they are the same strings,
   `'Different strings'` – otherwise.
7. Write a program that prints to the screen a word created by combining two strings provided by the user in such a way that it includes characters other than alphanumeric.
8. Write a program that prints to the screen a word created by removing the first occurrence of a string also provided by the user from the string provided by the user. For example, the result of running the program for the strings `"abrakadabra"` and `"ab"` will be the word `"rakadabra"`.
9. Write a program that prints to the screen a word created by removing from the string provided by the user all occurrences of the string also provided by the user. For example, the result of running the program for the strings `"abrakadabra"` and `"ab"` will be the word `"rakadra"`.
10. Write a program that creates a reversed word from the string provided by the user and prints it to the screen. For example, the result of running the program for the string `"hello"` should be the string `"olleh"` on the screen.
11. Write a program that prints to the screen the value `True` if the word provided by the user is a palindrome, and the value `False` otherwise. For example, as a result of running the program for the string `"level"` the value `True` should appear on the screen, while for the string `'label'` the value should be `False`.
12. Extend the above program to check whether the given sentence, after removing all spaces and changing the capital letters to lowercase, is a palindrome. For example, for the sentence `'A man, a plan, a canal - Panama'`, the program should print the value `True`.
13. Write a program that creates a word from a string provided by the user that is a concatenation (i.e. a combination) of this string and its inversion, then prints it to the screen. For example, as a result of running the program for the string `"link"`, the word `"linkknil"` should appear on the screen.
14. Write a program that encrypts the word provided by the user using a rail fence cipher with a fixed fence height. The program should also allow reading the encrypted text. The reader can find an explanation of what a rail fence cipher is, for example, at the link `https://en.wikipedia.org/wiki/Rail_fence_cipher`.
15. Write a program that, for the given strings representing numbers in the ternary system, displays their sum:
    - using the decimal system,
    - without using the decimal system.

16. Write a program that displays a natural number entered by the user in the Roman numeral notation (and vice versa).
17. Write a program that displays the `ASCII` codes of all characters from `a` to `z`.

Chapter 4

Functions

Python functions are used to create clear program code by dividing it into smaller fragments that can be reused as often as needed at any time. In general, a function in programming is understood in a broader sense than in mathematics. In mathematics, a function always has specific arguments and values. In programming, the set of arguments and the set of function values can be an empty set. This means that a function may not take any arguments, it may also not return any values, e.g. it can only be used to display a message.

This chapter will present the basics of creating and using functions, rules regarding the scope of variables, and passing arguments.

4.1. Function - general form

The `def` statement creates a function object and assigns it to a variable named the function name - a reference to the function object is assigned. The general format of a function definition is as follows:

```
def name(arg [[, arg1]...]):    #header
  instruction_block             #body
  [return statement]            #optional
```

A function definition consists of a function signature (header) - starting with the keyword `def`, followed by the function name and a list of arguments (sometimes called parameters). This list can be empty or contain one or more elements separated by a comma. The function header always ends with a colon. The body of a function is a single statement or block of statements that are executed when the function is called, indented the same depth at the beginning of each line. Every Python function has a return value. By default, the return value is `None`, unless a different value is returned from within the function using the `return` statement. The `return` statement can appear anywhere in the body of the function. When it is encountered, the function exits, returning the result to the caller. If the `return` statement is not present in the function code, the function exits with the last statement in the body and the control flow of the program moves outside the body.

Unlike other programming languages (like `C`), Python functions don't need to be fully defined before the program executes. `def` statements aren't evaluated until they're executed, and the body of the function isn't executed until after the function is called. Because `def` is a statement, it can be used anywhere statements appear—inside a conditional statement, an iteration statement, or even nested inside other `def` statements.

Python passes arguments to functions by assignment, which is an object reference. The default is positional passing, unless you specify otherwise. When a function is called, the values passed to the function default to the argument names in the function definition from left to right. In a function call, we can also pass arguments in the form of `name = value` pairs, or unpack any number of arguments using `*args` and `**kwargs` (for more information on this subject, the reader can refer to chapter 5).

According to the PEP8 documentation (https://peps.python.org/pep-0008/), which contains recommendations for making Python code clearer and more readable, a doc comment may or may not appear immediately after a function header.

Doc comments are usually enclosed in triple quotes (`"""`) so that they can span any number of lines.

```
def name(arg [[, arg1]...]):
  """ Optional single or multi-line doc comment """
  instruction_block
  [return statement]
```

Documentation comments can be entered in a variety of ways, but there are a few rules listed below:

- Triple quotes are used even when the comment is only one line long (see listing 4.1). This is a good practice for expanding the comment.
- No blank lines are left before or after the comment.
- A period is placed at the end of the comment.
- In the case of a multi-line comment describing a function (see listing 4.2):
  - The first line contains a short description of the function (line 2). The line ends with a period.
  - There is a blank line between the short and full description (line 3).
  - The full description (lines 4-15) contains a description of the function's parameters (arguments) and the function's return value. Each parameter (lines 10-13) and return value (lines 14-15) are described by name and type. This description also includes information on handling exceptional situations.

LISTING 4.1: *Singleline ducumentation comment*

```
def prime_numbers():
  """List of prime numbers in range of 1 to 100."""
  ...
```

LISTING 4.2: *Multiline documentation comment*

```
def give_account_number(pesel, last_name, name):
  """Identifies the customer's account number.

  The function give_account_number() returns the account number,
  using the given parameters.
  The pesel argument can only contain
  a number in the Universal Electronic
  Population Registration System (PESEL).
  An incorrect PESEL number will cause an exception to be thrown.
  :param pesel: customer's PESEL.
  :type pesel: str
  :param last name: Customer's last name.
  :type last name: str
  :return: Customer's account number.
  :rtype: str
  """
  ...
```

4.2. Function arguments

In the function header, you can define any number of arguments (formal parameters), separated by commas. In Python, there are four ways to define function arguments:

- positional arguments, required - these are arguments that do not have a default value assigned, they are at the beginning of the argument list;
- optional arguments, they are on the list after positional arguments, they have a default value;
- argument list `*args` - an arbitrary length list of unnamed arguments represented internally in the function as a tuple (more information about dictionaries can be found in the 5.2 section);
- argument dictionary `**kvargs` - an arbitrary length dictionary of named arguments, but not defined in the function, they must be placed after optional arguments (more information about dictionaries can be found in the 5.4 section).

If we specify arguments in the order they are defined in the header, the argument names can be omitted, but if we change their order or do not specify all optional arguments, their names should be specified. Names should be specified if we use `**kvargs`.

Arguments are passed to the function by automatically assigning objects to the names of local variables. To illustrate the operation of the argument passing feature, consider the example shown in listing 4.3. When the function is called on line 7, the object `-99` will be assigned to the variable `arg_f`, but `arg_f` exists only inside the called function, and its modification does not affect the place where the function was called - assignment inside the function changes the local variable `arg_f` to a completely different object (line 3). This is confirmed by the listed object identifiers illustrated in figure 4.1.

Listing 4.3: *Arguments vs. shared references*

```
1 def function(arg_f):
2   print('id(arg_f) >> before assigning:', id(arg_f))
3   arg_f = 111
4   print('id(arg_f) >> after assigning:', id(arg_f))
5 arg = -99
6 function(arg)
7 print(arg)
8 print('id(arg):',id(arg))
```

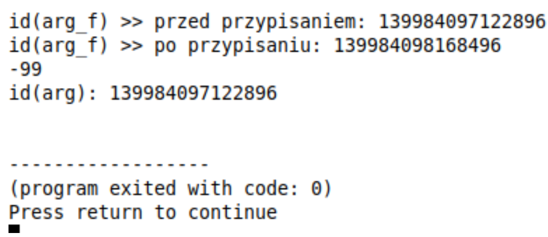

Figure 4.1: Result of executing program from listing 4.11

Argument names may initially share the object being passed, but only temporarily, the first time the function is called. After the argument name is reassigned in the function body, this relationship disappears.

However, when mutable objects are passed to the function, such as lists (subsection 5.1) or dictionaries (subsection 5.4), the modification of such objects in place may also exist after the function exits and thus have an impact on the calling code. Listing 4.4 shows the code in which the list `L` serves as both the input and output of the function, and the result of the assignment to `arg_f[1]` (line 2) in the function body affects the values of the list `L`, which is illustrated by listing its elements in the main function (line 6).

LISTING 4.4: *Mutable function argument*

```
def function(arg_f):
  arg_f[1] = 333
L = [-99, 0, 123, -11]
function(L)
print(L)
# OUTPUT
# [-99,333,123,-11]
```

You should be especially careful when working with arguments of mutable functions, and remember that you can always copy such an object within the function body so that any changes are made only locally to the copy within the function.

4.3. Function call

Function definitions do not change the program's control flow, i.e. the order in which the instructions are executed. The program always starts with its first instruction, and then each subsequent instruction is executed one at a time, in order from top to bottom. Instructions inside a function are not executed until the function is called. A function is called by its name and the current call parameters. If a function returns a value, we can use it in other instructions, including placing it in a variable by assigning it the return value. It can also be an argument to call another function or be a value printed to the standard output.

In a Python program, one of the functions has a special meaning. This is a function called `main`, which is the main function of the program and is responsible for controlling its operation. The Python interpreter, when processing source files, creates, among other things, a variable called `__name__`. If the source file is the main file (it is not loaded by another script, because then this variable takes the basic name of the script), then the variable `__name__` takes the value `"__main__"`. Using a conditional statement that checks the value of the `__name__` variable ensures that the `main` function is executed only if the script is the correct main script, so we can use it unchanged in other scripts. The 4.5 listing shows the skeleton of the `main` function along with its recommended call.

LISTING 4.5: *Calling the main function*

```
def main():
  """ The main function of the program """
  pass
if __name__ == "__main__":
  main()
```

Because functions divide a program into pieces of code, it is important to correctly place function definitions and the statements that call them. In listing 4.6 we show the incorrect order of statements because the Python interpreter will look for the definition of the `message` function above the call to the main function.

Listing 4.6: *Invalid order of instructions*

```
def main():
  print_message()
if __name__ == '__main__':
  main()
def print_message():
  print("*P*Y*T*H*O*N*")
```

Running the program from listing 4.6 will result in an error:

```
Traceback (most recent call last): ...
NameError: name 'print_message' is not defined
```

The correct order of defining functions in the program is shown in listing 4.7. In listing 4.8 we also have the correct, but also recommended order of defining functions, so that we always have the definition of the main function at the top of the screen.

Listing 4.7: *The correct order of instructions*

```
def print_message():
  print("*P*Y*T*H*O*N*")
def main():
  print_message()
if __name__ == '__main__':
  main()
```

Listing 4.8: *The correct and recommended order of instructions*

```
def main():
  print_message()
def print_message():
  print("*P*Y*T*H*O*N*")
if __name__ == '__main__':
  main()
```

In addition to arguments, which may be optional, every Python function has a return value. By default, the return value is `None`, but using the `return` statement in the function body allows you to return any expression:

```
return statement
```

Execution of the above instruction calculates the value of the expression `expression`, after which this value becomes the return value of the given function, which terminates its operation at this point. The return value can be a single value or a tuple of values and can be ignored by the function caller - it will then be rejected.

Listing 4.9 presents a program that calculates the area of a square with the side length provided by the user. The case when the data provided by the user is incorrect (the side length is zero or negative) (lines 3-7) has been taken into account. The value returned by the function calculating the area of the square (line 4) is assigned to a variable and used to print a message to the screen (line 5). In the definition of the function `area` the variable `result` was used, to which the value of the expression calculating the area of the square was assigned (line 10) and the value of this variable is returned by the function (line 11).

LISTING 4.9: *Function that calculates the area of a square*

```
1  def main():
2    b = int(input("Enter the length of the side of the square: "))
3    if b > 0:
4      fun = area(b)  #function call
5      print("Area: ", fun)
6    else:
7      print("Invalid input")
8
9  def area(a):
10   res = a*a
11   return res
12
13 if __name__ == '__main__':
14   main()
```

Listing 4.10 shows how you can use the `return` instruction property, which first calculates a value from an expression before returning it, to modify the definition of the function from listing 4.9 and shorten its code.

LISTING 4.10: *Correct function that calculates the area of a square*

```
def main():
  b = int(input("Enter the length of the side of the square: "))
  if b > 0:
    print("Area:",area(b))
  else:
    print("Invalid input")

```

```
def area(a):
  return a*a

if __name__ == '__main__':
  main()
```

In listing 4.11, we present the function returning more than one value - a tuple of values (lines 12 and 14). What is important, because it is a tuple, its elements do not have to be of the same type. The program first asks the user to enter the number of trials (line 2), and then in each iteration, the user enters two integers (lines 4 and 5), which the function `min_max` then returns in ascending order (line 6). The returned tuple is unpacked, and its elements are placed in appropriate variables. In this way, each of these variables can be used separately later in the program, if necessary, without having to refer to the tuple.

Listing 4.11: *A function that returns a pair of numbers in ascending order*

```
1  def main():
2   n=int(input("How many times do you want to make a comparison:"))
3   while n > 0:
4     a=int(input("a = "))
5     b=int(input("b = "))
6     min, max = min_max(a,b)
7     print("(", min, ",",max,")")
8     n -= 1
9
10 def min_max(a,b):
11  if a < b:
12    return (a,b)
13  else:
14    return (b,a)
15 main()
```

4.4. Practice exercises

1. Write a function `Print` that displays its single argument on the screen. Call it passing different types of arguments: string, integer, and floating point. Then call it without passing an argument. What happens then? And what happens when you pass two arguments?
2. Write a function that, based on the string given as an argument, returns a string in which all vowels in even-numbered positions are replaced with the character `'X'`.

3. Define and test a function `How_many_digits` in your program that, for a positive integer, given as an argument to the function, checks and returns the number of digits that the number consists of. Then, in the function `main`, call the function 10 times for a randomly generated argument value from the range $\langle 1, 50000\rangle$.
4. Declare and define a function `Inverse`, whose task is to return the inverse of an integer given as an argument. In the function `main`, calculate and print to the screen the sum of the reciprocals of 10 integers entered by the user that are not zero.
5. Define and call the function `Even`, which checks and returns `True` if the number given as an argument is even, `False` otherwise. In the function `main`, calculate and print to the screen how many even numbers there were among 100 integers generated randomly from the range $\langle 1, 100\rangle$.
6. Define a function `F1(a,b,c)` that calculates and returns how many numbers there are in the range $\langle a, b\rangle$ that are multiples of `c`. In the function `main`, read two integers `i` and `j`, from the standard input. If `j` is greater than `i`, read an integer from the standard input, the multiples of which will be counted by the function `F1` and print the value returned by the function to the screen. Otherwise, print the message "*Invalid data*" and terminate the program.
7. Define a function `F2(a,b)` that calculates and returns how many pairs of numbers `(x,y)` there are in the range $\langle a, b\rangle$ that satisfy the condition `3x=y`. In the function `F2` read two integers from the standard input `i` and `j` and if `j` is greater than `i` print the value returned by the function to the screen. Otherwise, print the message `'Incorrect data'` and terminate the program.
8. Define a function `F3(a,n)` that will calculate and return the value of the expression:
$$\frac{a}{a+1} - \frac{a+1}{a+2} - \frac{a+2}{a+4} - \frac{a+3}{a+8} - \ldots - \frac{a+n}{a+2^n}$$
In the function `main`, read two integers `a` and `n` from the standard input 10 times, if they are positive, print the value returned by the function `F3` to the screen. Otherwise, print the message `'Incorrect data'` and terminate the program.
9. Define a function `F4(a,n)` that will calculate and return the value of the expression:
$$\frac{1}{a} + \frac{1}{a+2} + \frac{1}{a+4} + \frac{1}{a+6} + \ldots + \frac{1}{a+2\cdot n}$$
In the function `main`, read two integers `a` i `n` i from the standard input 10 times, if they are positive, print the value returned by the function `F4` to the screen. Otherwise, print the message `'Incorrect data'` and terminate the program.
10. Define a function `F5(n)` that prints to the screen all Pythagorean triples (i.e. triples of integers `a, b, c` such that $a^2 + b^2 = c^2$), consisting of numbers less than `n`. In the function `main`, call the function `F5` for a positive integer `n` supplied by the user.
11. Write a function that, for the string given as an argument, returns the most common vowel in it.

12. Define a function called `dice_throw` that simulates the throw of a six-sided die, i.e. the function should return a random integer from the mutually closed range from 1 to 6. Using this function, program a simple game (with two players) in which each player throws two dice in each round (i.e. calls the function `dice_throw` twice). The player who rolls the highest number wins the round. The game is won by the player who has more victories after `n` turns. If there is a tie after `n` turns, the game continues until the first victory. The computer is to play itself, and additionally, after each turn, it will print on the screen the number of points drawn and the current balance of victories and defeats. The number `n` is provided by the user.
13. Write a program that calculates the sum of the expression
$$1 + 2 - 3 + 4 - 5 + ... \pm n$$
for any `n`.
14. Write a program that calculates the sum of the expression
$$1 * 2 + 2 * 3 + 3 * 4 + ... + n * (n + 1)$$
for any `n`.
15. Write a function that accepts one argument - the number of teams in a football league. The function's task is to return information about how many matches should be played in this league in a round robin system (without return matches).
16. Define the function `F6(a,b,c)` that checks whether a triangle can be built from sides with lengths: `a`, `b`, `c`.
17. Define a function that accepts as an argument - the speed expressed in meters per second. The function's task is to return the speed expressed in kilometers per hour.
18. Define a function that checks whether the number given as an argument to the call is a perfect number.
19. Define a function that accepts one argument - the percentage score obtained on the colloquium. The function's task is to return the grade corresponding to the given result.
20. Define a function that accepts one argument - a positive natural number n. The function's task is to calculate and return the sum of the first n terms of a sequence with the general formula:
$$a_n = \frac{(-1)^{n+1}}{(n+1)n}$$
.
21. Write and test the operation of the function `Caesar(napis)`, which encrypts a given string with the Caesar Cipher (`https://en.wikipedia.org/wiki/Caesar_cipher`).
22. Write a function that returns the number of consonants in the string given as an argument to the function.

23. Write and test the operation of a function that accepts one argument, which is a string. The function's task is to return a string in such a way that the first character, if it is a lowercase letter, is converted to an uppercase letter, e.g. for the string `'spring'` the function will return the string `'Spring'`.
24. Write and test the function `F7(string1,string2)` that checks whether the same characters are in the appropriate positions in both strings. The function should return a number corresponding to the number of differences (or 0 if the strings are identical).
25. Write a function `F8(string1,string2,string3)` that returns a string that is a concatenation of the three strings given (use alphabetical order for concatenation).
26. Write a function `F9(string1,string2)` that checks whether the characters in the first string can be used to create a second string.

Chapter 5

Data Structures

There are four main data structures in Python: lists, tuples, sets, and dictionaries, which differ in the way they store and manage data, including their approach to duplicates. In this chapter, we will discuss each of the above data structures in terms of their structure and applications.

5.1. Lists

The most useful type for grouping different values is `list`, which can be written as a sequence of elements separated by commas, enclosed in square brackets.

5.1.1. Lists - basic information

When creating a list, we don't have to worry about its type uniformity because in Python, the elements of a list don't have to be of the same type, e.g.

```
>>> shopping_list = ['bread', 1.95, 'milk', 2.19, 'butter', 3.20]
```

List elements are indexed, i.e. the elements are assigned ordinal numbers starting from 0 for the first element in the list, and their value increases by one with each subsequent element in the structure. Indexes allow you to extract from the list the value contained in it at that index. You can also use negative integers to index list elements, e.g. for the list `a=[2,3,4,5,10]`, its elements can be indexed by numbers from `-5` to `-1`.

```
>>> print(shopping_list[0], shopping_list[2], shopping_list[4])
chleb mleko masło
```

Lists can be subject to splitting and merging operations, e.g.

```
>>> shopping_list = ['bread', 1.95, 'milk', 2.19, 'butter', 3.20]
>>> shopping_list[1:-1]
[1.95,'milk',2.19,'butter']
>>> shopping_list[:2] + ['lettuce', 2 * 2.2]
['bread',1.95,'lettuce',4.4]
>>> 2 * shopping_list[:3] + ['end']
['bread',1.95,'milk','bread',1.95,'milk','end']
```

A list is a mutable structure in which means all of its individual elements can be changed, e.g.

```
>>> shopping_list = ['bread', 1.95, 'milk', 2.19, 'butter', 3.20]
>>> shopping_list[1] += 0.23
>>> shopping_list
['bread', 2.18, 'milk', 2.19, 'butter', 3.20]
>>> arr = ['a', 'b', 6, 12]
>>> arr[0:2] = [1, 2]     # Replaces some elements
>>> arr
[1, 2, 6, 12]
>>> arr[0:2] = []         # Deletes some elements
>>> arr
[6, 12]
>>> arr[1:1] = ['c', 'd'] # Inserts some elements
>>> arr
[6, 'c', 'd ', 12]
>>> arr[:0] = arr         # Inserts list to beginning
>>> arr                   # (in this case we duplicate its elements)
[6, 'c', 'd ', 12, 6, 'c', 'd', 12]
>>> arr[:] = []           # Removes all elements of list
>>> arr
[]
```

Listing 5.1 shows sample code for a program that creates a list of the squares of the natural numbers 1 through 20 and then displays them on the screen in reverse order. Adding elements to the initially empty list is done using `append`, which is a method of the `list` class. The elements displayed on the screen are preceded by the index at which this value is located in the list. For this purpose, we use the `enumerate` function, which adds a counter to each element of the iterable object and returns the enumerate object, which is a tuple of `(index, value)`, as an output.

LISTING 5.1: Append *method example*

```
a = list()                         #declare an empty list
for i in range (20):
  a.append((i+1)**2)               #add more elements
print(a)                           #display the list
for i, v in enumerate(a):
  print(i,':',v,", ",end="")       #display pairs (index,element)
print()
```

Lists can be concatenated together using the concatenation operator (+), so the value being added to the list must be enclosed in square brackets, as shown in listing 5.2. You can also use a shortcut (a+=i+1,) since adding a comma already forces the Python interpreter to treat a single element as a single-element sequential structure (e.g. a list). To list the elements of a list, we use the `enumerate` function again, this time with a second optional argument specifying the initial value for the counter.

LISTING 5.2: *Merging lists*

```
a = list()                         #declare an empty list
for i in range (20):
  a += [(i+1)**2]                  #add more elements
  #a += (i+1)**2,                  #alternative way of adding elements
print(a)                           #display the list
for i, v in enumerate(a,1):
  print(i,':',v)
print()
```

Listing 5.3 presents a program that fills a 10-element list with real numbers entered by the user and calculates their sum.

LISTING 5.3: *Sum of list elements*

```
a = list()
for i in range (10):
  f = float(input("Enter list element:"))
  a += f,
print(a)
total = 0
for i in a:
  total += i
print("The total sum of all elements is:", total)
```

Listing 5.4 shows a modified program for calculating the sum of list elements, which does this in the same loop that creates the list from numbers entered by the user.

LISTING 5.4: *Sum of list elements using one loop*

```
a = list()
total= 0
for i in range(10):
  a.append(float(input("Enter "+str(i + 1)+" element:")))
  total = total + a[i]
print("Your list:\n", a)
print("The sum of elements is:", total)
```

Listing 5.5 presents a program that fills a 10-element list with random integer numbers from the interval `<1,10>` and calculates the product of the list elements.

LISTING 5.5: *Product of elements in the list*

```
import random
a = list()
for i in range (10):
  a += random.randint(1,10),
product = 1
print("Your list: ", a)
for i in range(len(a)):
  product *= a[i]
print("product: ", product)
```

Listing 5.6 presents a modified program that calculates the product of list elements in the same loop in which the list is created from randomly generated numbers.

LISTING 5.6: *Product of elements in the list using only one loop*

```
import random
a = list()
product = 1
for i in range(10):
  a.append(random.randint(1,10))
  product *= a[i]
print("Your list: ", a)
print("product:", product)
```

The built-in functions **len**, **min**, **max**, and **sum** can be used with lists.

```
>>> arr = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
>>> len(arr)
12
>>> min(arr)
28
```

```
>>> max(arr)
31
>>> sum(arr)
365
```

It is possible to **nest** lists, i.e. create lists whose elements are other lists.

```
>>> b = [2, 3]; a = [1, b, 4]
>>> a
[1, [2, 3], 4]
>>> len(a)
3
>>> len(a[1])
2
>>> a[1]
[2, 3]
>>> a[1][0]
2
```

You can remove a list element at a given index, as well as a list slice, using the `del` instruction.

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

5.1.2. Chosen `list` class methods

- `append(value)` - appends an item `value` at the end of the list.

  ```
  >>> a = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
  >>> a.append([4, 4, 4])
  >>> a[0].append("Python")
  >>> a
  [[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
  ```

- `extend(iterable)` - adds the elements of the iterable object `iterable` to the list.

```
>>> a = [3, 6, 8]
>>> a.extend([3, 7, 6])
>>> a
[3, 6, 8, 3, 7, 6]
```

- `insert(index,value)` - inserts the object `value` into the list at the location specified by `index`.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.insert(3, 123)
>>> a
[3, 6, 8, 123, 3, 7, 6]
```

- `remove(value)` - removes the first occurrence of the `value` object from the list. If the `value` object is not in the list, it throws a `ValueError` exception.

```
>>> a = [3, 6, 8, 123, 3, 7]
>>> a.remove(3)
>>> a
[6, 8, 123, 3, 7]
```

- `clear()` - removes all elements from the list.

```
>>> a = [3, 6, 8, 123, 3, 7]
>>> a.clear()
>>> a
[]
```

- `pop(index=-1)` - removes an item from the given list position (defaults to the last one). Throws an exception `IndexError` if the list is empty or the index is out of range.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.pop()
6
>>> a
[3, 6, 8, 3, 7]
>>> a.pop(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

- `index(value,start=0,stop=9223372036854775807)` - returns the index of `value` in the list, optionally in the range `<start,stop)`. If `value` is not in the list, it throws a `ValueError` exception.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.index(7)
4
>>> a.index(8,3,5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 8 is not in list
```

- `count(value)` - returns the number of occurrences of `value` in the list.

```
>>> a = [3, 6, 8, 3, 7, 6]
>>> a.count(6)
2
>>> a.count(0)
0
```

- `reverse()` - reverses the order of elements in the list.

```
>>> a = [123, 8, 7, 6, 3]
>>> a.reverse()
>>> a
[3, 6, 7, 8, 123]
```

- `sort(key=None,reverse=False)` - sorts the list in non-decreasing order and returns `None`. The sorting is stable, i.e. the order of equal elements is preserved. If the `key` function is given, it is applied to each element of the list and sorted according to the values of the `key` function. The `reverse` flag can be set to sort in non-ascending order.

```
>>> a = [6, 8, 123, 3, 7]
>>> a.sort()
>>> a
[3, 6, 7, 8, 123]
>>> a.sort(reverse = True)
>>> a
[123, 8, 7, 6, 3]
>>> a = [3, 7, 2, -3, 9, 7, -4, -8, -5, 8]
>>> a.sort()
```

```
>>> a
[-8, -5, -4, -3, 2, 3, 7, 7, 8, 9]
>>> a.sort(reverse=True)
>>> a
[9, 8, 7, 7, 3, 2, -3, -4, -5, -8]
>>> a.sort(key=abs)
>>> a
[2, 3, -3, -4, -5, 7, 7, 8, -8, 9]
>>> a.sort(key=abs, reverse=True)
>>> a
[9, 8, -8, 7, 7, -5, -4, 3, -3, 2]
>>> a = ['Adam','Eva', 'Mark', 'Will']
>>> def F(L):
...     return len(L)
...
>>> a.sort(key = F)
>>> a
['Eva', 'Adam', 'Mark', 'Will']
>>> a.sort(key = F, reverse = True)
>>> a
['Will', 'Mark', 'Adam', 'Eva']
>>> a.sort(key = len)
>>> a
['Eva', 'Adam', 'Mark', 'Will']
```

- `copy()` - creates a shallow copy of a list. A shallow copy creates a new list, and then (if possible) inserts references to objects found in the original.

```
>>> a = [3, 6, 8]
>>> a.extend([3, 7, 6])
>>> b = a.copy()
>>> b
[3, 6, 8, 3, 7, 6]
>>> id(a) == id(b)
False
>>> old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> new_list = old_list.copy()
>>> old_list.append([4, 4, 4])
>>> old_list
[[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
```

```
>>> new_list
[[1, 1, 1], [2, 2, 2], [3, 3, 3]]
>>> old_list[0].append("Python")
""" The change of the element with index 0 in old_list
    is also visible in new_list"""
>>> old_list
[[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
>>> new_list
[[1, 1, 1, 'Python'], [2, 2, 2], [3, 3, 3]]
""" Because new_list[0] is a reference to old_list[0]"""
>>> id(old_list[0]) == id(new_list[0])
True
```

In listing 5.7, we show the time difference of creating a list with three different operations: expanding `extend`, copying `copy`, and cutting `[:]`. To calculate the running time of each of them we use the `time` function from the `time` module.

Listing 5.7: `copy` method example

```python
from time import time

old_list = 10**7 * [2, 3, 5, 7, 11]
t = time()
new_list = [].extend(old_list)
print("list.extend: ", time() - t)

t = time()
new_list = old_list.copy()
print("list.copy: ", time() - t)
t = time()
new_list = old_list[:]
print("list slicing [:]:", time() - t)
```

The program execution result confirms that the `copy` method is the fastest in its operation:

```
list.extend:  0.8033039569854736
list.copy:  0.6313166618347168
list slicing [:]: 0.8029310703277588
```

Listing 5.8 shows the creation of a list from an existing list, which in effect leads to the creation of a list `B` which is a reference to the object `A`.

Listing 5.8: *Matrix example*

```
A = [0,1]
B = []
for i in range(2):
    B.append(A)
print('A[0]:',id(A[0]))
print('B[0][0]:',id(B[0][0]))
print('B[1][0]:',id(B[1][0]))
print(B)
A[0]=100
print(B)
```

This is confirmed by the program's output, which prints the `id` of the zeroth element of the list `A` and the zeroth elements of the rows of the matrix `B`. Additionally, since the matrix `B` was created as a list of two references to the object `A`, any modification of the elements of the object `A` is also visible in the matrix `B`:

```
A[0]: 140643489186000
B[0][0]: 140643489186000
B[1][0]: 140643489186000
[[0, 1], [0, 1]]
[[100, 1], [100, 1]]
```

5.1.3. Program call arguments

The built-in module `sys` contains commands related to Python and its environment. It allows handling of arguments for program invocation from the command line, which allows you to create programs even more flexibly tailored to the needs of the user. The variable `sys.argv` is a list of strings that are arguments with which the program was invoked from the command line. Where the first element `argv[0]` is the name of the Python script and, depending on the operating system, it may be a full path or not. If the command was executed using the -c command line option of the interpreter, then `argv[0]` is set to the string `'-c'`. If no script name was passed to the Python interpreter, then `argv[0]` is an empty string.

Listing 5.9: *Printing program call arguments as a list*

```
import sys
def main():
  print(sys.argv)
if __name__ == "__main__":
  main()
```

As a result of calling the program from listing 5.9, the contents of `sys.argv` will be printed to the screen.

```
~$ python3 main_arg_1.py 1 "Alice" [1,2,3]
['main_arg_1.py', '1', 'Alice', '[1,2,3]']
```

Another way to list command line arguments, illustrated in listing 5.10, is to use a `for` loop and display each element of the `sys.argv` list separately.

LISTING 5.10: *Listing program call arguments on separate lines*

```
import sys
def main():
  for a in sys.argv:
    print(a)
if __name__ == "__main__":
  main()
```

As a result of calling the program from listing 5.10, the contents of `sys.argv` will be printed to the screen.

```
~$ python3 main_arg_2.py 1 "Alice" [1,2,3]
main_arg_2.py
1
Alice
[1,2,3]
```

And yet, a third way to list the arguments of the program call from listing 5.11 using a `for` loop, but using the `range` function and indexing.

LISTING 5.11: *Listing the arguments of the program call using the* `range` *function*

```
import sys
def main():
  for j in range(len(sys.argv)):
    print(sys.argv[j])
if __name__ == "__main__":
  main()
```

As a result of calling the program from listing 5.11, the contents of `sys.argv` are printed to the screen.

```
~$ python3 main_arg_3.py 1 "Alice" [1,2,3]
main_arg_3.py
1
Alice
[1,2,3]
```

5.2. Tuples

A tuple is a sequence whose contents cannot be changed (an unchangeable sequence). When creating a tuple, round brackets are used, for example:

```
t = (1, 2, 3, 4,) # Four-element tuple
```

The tuple values are separated by commas. An empty tuple has the form: `t = ()` while a tuple with one element has the form: `t = (1,)`. Note that the comma in a single-element tuple is mandatory. Without the comma, it is not a tuple object, but an integer. In the case of multi-element tuples, the comma is optional.

Since a tuple is an unchangeable sequence, it "supports" all those operations that do not change its contents, i.e.:

- indexing (only for retrieving element values);
- methods such as `index()`;
- built-in functions such as `len()` , `min()` and `max()`;
- slice expressions;
- operator `in`;
- concatenation operators `+` and multiplication operators `*`.

A tuple is an iterable object, so we can use a `for` loop to iterate through its elements. For example, let's print the names contained in a tuple named `names` using a `for` loop and indexing.

```
names = ('Jacob','Billy','Max','Frankie',)
for n in range(len(names)):
  print("names[", n, "] = ",names[n],".",sep = "")
```

Tuples can be combined with each other and duplicated, e.g.

```
t1 = (1, 2, 3, 4,)
t2 = (10, 20, 30, 40,)

print("t1 = ", t1, ".", sep = "")
print()
print("t2 = ", t2, ".", sep = "")
print()
t = t1 + 3*t2
print("t1 + 3*t2 = ",t, ".",sep = "")
```

Listing 5.12 presents the code of a program that, using the built-in functions `min` and `max`, searches for and prints the value of the smallest and largest elements of a tuple and the indexes under which they are located.

Listing 5.12: *Looking for* `min` *and* `max` *elements in a tuple*

```
t1 = (11, 2, 332, 4)
t2 = (1, 20, 30, 40)

print("t1 = ", t1, ".", sep = "")
print("t2 = ", t2, ".", sep = "")
print()
t = t1 + t2
print("t = ", t, ".", sep="")
mn = min(t)
mx = max(t)
print("min:t[",t.index(mn),"]=",mn,".",sep="")
print("max:t[",t.index(mx),"]=",mx,".",sep="")
```

Tuples are much faster to compute than lists. We use them when we are dealing with elements that should not be subject to modification, such as a list of days of the week or a calendar date. Tuples provide security that none of their elements, as well as themselves, will change during the operations performed. We use them to store heterogeneous data types. In such cases, we say that we are dealing with heterogeneous types. Tuples can be nested within each other, and referencing a single element in a nested tuple requires the use of an additional indexing operator, for example, in the listing 5.13, a tuple is defined, which consists of two nested tuples. In the reference `Tuple[2]`, the second element of the tuple, the one with index 2, which is also a tuple, is identified. So, to show its last element, which is the string `Hi`, we need to use the indexing operator again and the position number where this string is located, i.e. `Tuple[2][2]`.

Listing 5.13: *Nested tuples*

```
Tuple = ("Python", (2, 4, 6), (4, 5.6, "Hi"))

print("Tuple contents:", Tuple)
print("Firste lement of Tuple:")
print("\t* First:", Tuple[0])
print("\t* Second:", Tuple[1])
print("\t* Third:", Tuple[2])
print("\t* Last:", Tuple[-1])
print("\t* First two:", Tuple[:-1])
print("\t* First in last element:", Tuple[-1][0])
print("\t* Last in last element:", Tuple[-1][-1])
print("\t* Last two in second element:", Tuple[1][1:])
```

One use of a tuple is to use it as an optional special argument to a function marked `*args`. The form `*args` specifies that any additional unnamed positional arguments of the call are gathered in the tuple and are associated with the name **args**. With this special argument, we can create functions with any number of arguments, including a function with no arguments.

Listing 5.14 shows a practical use of the special argument, which gathers integers passed as arguments in the tuple **args** to calculate and return their sum. If the function is called without arguments, the function returns zero.

Listing 5.14: *Any number of unnamed positional arguments*

```
def main():
  print(func())
  print(func(3))
  print(func(3, 5))
  print(func(31, 28, 31))
  print(func(3, 5, 31, 28))
  print(func(3, 5, 8, 31, 28))

def func(*args):
  _sum = 0
  if len(args) != 0:
    _sum = sum(args)
  return _sum

if __name__=='__main__':
  main()
```

The special argument **args** appears in the function header after the positional arguments and optional positional arguments. The name **args** is a common abbreviation for "*arguments*" and is usually a variable that contains a tuple of positional arguments.

In listing 5.15, we show the general form of a function definition, in which the first two arguments are positional, the second of which is optional due to the default value assigned to it, and the third is a special argument. Calling the function with one argument (line 2) will cause the parameters of the current call to be matched from the left to the arguments, and so the argument `a` will take the value 1, the argument `b` will have its default value set, and the special argument **args** will be empty. In the case of a function call with two arguments (line 3), the first two function arguments will take the values of the current parameters in the order they appear from the left, and the special argument will still be an empty tuple. Each call with more than two formal parameters will cause them to be collected in the special argument tuple, starting from the third parameter of the call.

Listing 5.15: *General form of a function definition with unnamed arguments*

```python
def main():
  func(1)
  func(1, 2)
  func(1, 2, 3, 4, 5)

def func(a, b = 9, *args):
  # a is a mandatory positional argument
  print("a =", a)
  # b is an optional positional argument
  print("b =", b )
  # args is a tuple of unnamed
  # positional arguments
  print("args =", args)

if __name__=='__main__':
  main()
```

As a result of executing listing 5.15 the screen will display:

```
a = 1
b = 9
args = ()
a = 1
b = 2
args = ()
a = 1
b = 2
args = (3, 4, 5)
```

Attempting to call the `func` function without arguments will result in an error:

`TypeError: func() missing 1 required positional argument: 'a'.`

5.3. Sets

Python has a built-in data type for sets. *A set* is an unordered collection that contains no duplicates.

5.3.1. Sets - the basics

Using the `in` operator, you can test whether an element belongs to a set. Sets allow you to perform basic operations such as union, intersection, difference, and symmetric difference. To create a set, use the {} brackets for a sequence of objects separated by commas or the `set` function with an iterable object as an argument. However, an empty set is created only using the `set` function without arguments, not the {} brackets, because the {} construction creates an empty dictionary.

In listing 5.16, we present different ways of creating sets: by enclosing the names of fruits in curly brackets (lines 2 and 3), by calling the `set` function for arguments that are strings (lines 8 and 9), or lists of numbers (lines 17 and 18) and the basic operations performed on these sets: sum (`|`), difference (`-`), product (`&`) and symmetric difference (`^`). The operations listed are consistent with the operations on sets known from set theory.

Listing 5.16: *Collections and operations on them*

```
1  def main():
2    basket = {"apple", "kiwi", "apple",\
3    "lemon", "kiwi", "banana", "mango"}
4    print(basket)
5    print("kiwi" in basket) # True
6    print("mandarin" in basket) # False
7    a = set("abracadabra")
8    b = set("Alicecazam")
9    print("a =", a)
10   print("b =", b)
11   print("a - b =", a - b) # difference of sets
12   print("a | b =", a | b) # union of sets
13   print("a & b =", a & b) # product of sets
14   print("a ^ b =", a ^ b) # symmetric difference of sets
15   a = set([1,2,3,4,5,6,7,8,9,0])
16   b = set([1,1,2,2,3,3,4,4,5,5,9,9])
17   print("a =", a)
18   print("b =", b)
19   print("a - b =", a - b) # difference of sets
20   print("a | b =", a | b) # union of sets
21   print("a & b =", a & b) # product of sets
22   print("a ^ b =", a ^ b) # symmetric difference of sets
23 if __name__ == "__main__":
24   main()
```

As a result of executing the program from listing 5.16 we will see on the screen:

```
{'lemon', 'banan', 'jabłko', 'mango', 'kiwi'}
True
False
a = {'a', 'c', 'r', 'b', 'd'}
```

```
b = {’m’, ’a’, ’c’, ’z’, ’l’}
a - b = {’d’, ’b’, ’r’}
a | b = {’m’, ’a’, ’c’, ’r’, ’z’, ’l’, ’b’, ’d’}
a & b = {’a’, ’c’}
a ^ b = {’z’, ’l’, ’b’, ’m’, ’d’, ’r’}
a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
b = {1, 2, 3, 4, 5, 9}
a - b = {0, 8, 6, 7}
a | b = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
a & b = {1, 2, 3, 4, 5, 9}
a ^ b = {0, 6, 7, 8}
```

In turn, in listing 5.17, we present again a basket of fruits represented by a set of their names (lines 2 and 3), which is printed to the screen and then converted to a list (line 5). The elements of the list are printed to the screen (line 6) to see (lines 6 and 8) their order before and after sorting (line 7). From the set named `basket`, all its elements are removed using the `clear` method in order to insert each element of the list into the set `basket` using the `add` method in the `for` loop (lines 10 and 11). After printing the elements of the set to the screen, it turns out that their order is different from the order of the elements of the list inserted into the set arranged in ascending order. In Python, sets are immutable and unordered collections.

Listing 5.17: *Zbiory i działania na nich*

```
1  def main():
2    basket = {"apple", "kiwi", "apple",\
3    "lemon", "kiwi", "banana", "mango"}
4    print("1:", basket)
5    arr = list(basket)
6    print("2:", arr)
7    arr.sort() # sort list
8    print("3:", arr)
9    basket.clear()
10   for fruit in arr:
11     basket.add(fruit)
12   print("4:", basket)
13   basket.clear()
14
15 if __name__ == "__main__":
16   main()
```

As a result of executing the program from listing 5.17 the following information will appear on the screen:

```
1: {'banana', 'lemon', 'mango', 'kiwi', 'apple'}
2: ['banana', 'lemon', 'mango', 'kiwi', 'apple']
3: ['banana', 'lemon', 'apple', 'kiwi', 'mango']
4: {'banana', 'lemon', 'mango', 'kiwi', 'apple'}
```

5.3.2. Chosen `set` class methods

In this subsection, we will only list the methods of the `set` class without providing examples of their use. We encourage the reader to test them in programs that implement the content of the tasks to be solved independently.

- `a.add(x)` – Adds the object `x` to the set `a`.
- `a.clear()` – Removes all elements from the set `a`.
- `a.copy()` – Returns a shallow copy of the set `a` (copies only references to objects that are elements of the set).
- `a.discard(elem)` – Removes an element from the set `a`, if it belongs to that set. Otherwise, it does nothing.
- `a.difference(*others)` – `a - other_1 - other_2 - ...`
  Returns the difference of two or more sets as a new set.
  (The new set will contain all elements of set `a`, that do not belong to the other sets.)
- `a.difference_update(*others)` – `a -= other_1 | other_2 | ...`
  Removes all elements from the set `a` that belong to the other sets.
- `a.intersection(*others)` – `a & other_1 & other_2 & ...`
  Returns the intersection of the set `a` and the other sets as a new set.
- `a.intersection_update(*others)` – `a &= other_1 & other_2 & ...`
  Modifies the set `a` retaining only those elements that also belong to the other sets.
- `a.isdisjoint(other)` – `a != other`
  Returns `True` if both sets are disjoint.
- `a.issubset(other)` – `a <= other`
  Returns `True` if the set `a` is contained in the set `other`.
- `a.issuperset(other)` – `a >= other`
  Returns `True` if the set `a` contains the set `other`.
- `a.pop()` – Removes one element from the set `a` and returns it as the value of the method. Throws a `KeyError` exception if the set is empty.
- `a.remove(elem)` – Removes the specified element from the set `a`. If the element is not in the set, throws a `KeyError` exception.

- `a.symmetric_difference(other)` – `a ^ other`
  Returns the symmetric difference of two sets as a new set.
- `a.symmetric_difference_update(other)` – `a ^= other`
  Modifies the set `a` as the symmetric difference of this set and the set `other`.
- `a.union(*others)` – `a | other_1 | other_2 | ...`
  Returns the union of the set `a` and the remaining sets as a new set.
- `a.update(*others)` – `a |= other_1 | other_2 | ...`
  Modifies the set `a` as the union of this set and the remaining sets.

5.3.3. Frozen sets

In Python, there is a special type of set called `frozen set`, which is an object of the class `frozenset`. Frozen sets are sets that cannot be modified: neither can new elements be added to them, nor elements removed from them. They can be elements of other sets, as well as of frozen sets, because they are both *immutable* and *hashable*. Frozen sets can be created using the function `frozenset`, whose argument can be any iterable object. Calling the function `frozenset` without arguments creates an empty frozen set. The following methods are available for frozen sets: `copy`, `difference`, `intersection`, `isdisjoint`, `issubset`, `issuperset`, `symmetric_difference` and `union`.

We will now present some examples of operations on two frozen sets created from list elements (sequences):

```
>>> A = frozenset([1,2,3,4])
>>> B = frozenset([1,2,5,6])
```

- Checking whether the sets `A` and `B` are disjoint: `isdisjoint`:

```
>>> A.isdisjoint(B)
False
```

- Calculating the difference of sets `A` and `B`:

```
>>> A.difference(B)
frozenset({3, 4})
```

- Calculating the union of sets `A` and `B`:

```
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
```

- Calculating the symmetric difference of sets `A` and `B`:

```
>>> A ^ B
frozenset({3, 4, 5, 6})
```

- Calculating the intersection of sets `A` and `B`:

```
>>> A & B
frozenset({1, 2})
```

Attempting to add an element to a frozen set that is immutable results in the error `AttributeError`:

```
>>> A.add(11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Since frozen sets are hashable, you can use the `hash` function on them:

```
>>> hash(A)
5575258175646371796
>>> hash(B)
2799569190866510694
```

5.4. Dictionaries

Another one of the built-in types in Python is `dictionary`. They define the mutual relation between a key and a value. A dictionary is a mutable (modifiable), unsorted set of `key:value` pairs enclosed in curly braces (`{}`). Dictionaries can be compared to "associative memory" or "associative arrays" from other programming languages or the primary key in revational databases, which allows you to identify a single row in a table. Dictionaries are internally implemented as hash tables, additionally optimized, which makes data retrieval from them very fast.

Indexed dictionaries are keys that must be immutable objects, such as numbers or strings. Tuples can also be used as keys if their components are numbers, strings, tuples, and frozen sets. You cannot use regular lists as keys because they can be modified, for example, using the `append` method.

5.4.1. Creating a dictionary

Creating an empty dictionary requires either assigning empty curly braces to the `d` variable:

```
>>> d = {}
>>> d
{}
```

or using the built-in function `dict`:

```
>>> d = dict{}
>>> d
{}
```

You can create a dictionary with elements that are `key:value` pairs separated by commas and delimited by curly braces, e.g.

```
>>> d = {"a":"Alice", "b":"Beatrice"}
>>> d
{'a': 'Alice', 'b': 'Beatrice'}
```

In this case, both keys and values are strings.

Now we will use the `dict` function to get the same dictionary.

```
>>> d = dict("a"="Alice", "b"="Beatrice")
>>> d
{'a': 'Alice', 'b': 'Beatrice'}
```

Listing 5.18 shows different ways of creating dictionaries:

- in the standard way - lines 2 and 3;
- using the `zip` function (line 4) - this function combines elements from different iterable objects, such as lists, tuples, sets, and returns a `iterator` (more information about iterators can be found in the 8.4 subsection). We can use it to combine two lists to create pairs that are inserted into the dictionary;
- using a list of two-element tuples - line 5;
- using a different dictionary - line 6.

Listing 5.18: *Creating dictionaries*

```
1  def main():
2    a = dict(one=1, two=2, three=3)
3    b = {'one': 1, 'two': 2, 'three': 3}
4    c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
5    d = dict([('two', 2), ('one', 1), ('three', 3)])
6    e = dict({'three': 3, 'one': 1, 'two': 2})
7    print(a, b, c, d, e, sep='\n')
8    print(a == b == c == d == e)
9  if __name__=='__main__':
10   main()
```

An element in a dictionary is identified by a `key`. So, to get the value of an element, you need to use square brackets for the dictionary variable and inside it, you need to specify the key that identifies the element:

```
>>> d["a"]
'Alice'
>>> d["b"]
'Beatrice'
```

You can access the value using the key, but you can't access the key using the value. So `d['b']` returns `'Beatrice'`, but calling `d['Beatrice']` will throw an exception because `'Beatrice'` is not a key of the dictionary `d`.

```
>>> d["Alice"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Alice'
```

5.4.2. Adding and overwriting data in a dictionary

Since keys in a dictionary cannot be repeated, when assigning a value to an existing key we will be overwriting the older value.

```
>>> d['b'] = 'barbara'
>>> d
{'a': 'Alice', 'b': 'barbara'}
```

At any time, we can add a new `key:value` pair to the dictionary. The syntax for adding a pair to the dictionary is identical to that for modifying an existing value. Unfortunately, this can sometimes cause a problem, because we may think we've added a new value to the dictionary when in fact we've overwritten an existing one.

```
>>> d['c'] = 'Chase'
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 'Chase'}
```

Note that the dictionary does not store keys in a sorted manner.

Since Python 3.6, the order of `key:value` pairs is the order in which they were inserted into the dictionary (in previous versions of Python, the order of entries was unpredictable). Note that key names are case-sensitive, as we show in the following example.

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "another value"
>>> d
{'key': 'another value'}
>>> d["Key"] = "another value"
```

```
>>> d
{'key': 'another value', 'Key': 'another value}
```

Dictionaries aren't just for strings. A value in a dictionary can be any data type: a string, an integer, an object, or even another dictionary. In a single dictionary, all the values don't have to be the same type; we can put objects of mixed data types into it, such as strings and integers.

```
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 'Chase'}
>>> d["d"] = 3
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 3}
```

The keys in a dictionary are more restrictive, but can be strings, integers, and a few other types that are immutable. However, keys within a dictionary do not have to be the same type, e.g. they can be strings as well as integers.

```
>>> d[42] = "Will"
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 3, 42: 'Will'}
```

5.4.3. Deleting elements from a dictionary

To remove a specific element from the dictionary that is pointed to by the given key, we use the `del` instruction.

```
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 3, 42: 'Will'}
>>> del d[42]
>>> d
{'a': 'Alice', 'b': 'Beatrice', 'c': 3}
```

Attempting to delete an item referenced by a key that is not in the dictionary results in the error `KeyError`.

```
>>> del(d[42])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 42
```

5.4.4. Dictionary inbuilt methods

The basic operations performed on dictionaries include:

- Checking if a key is in the dictionary using the operator `in`.

```
>>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
>>> 5 in Dict
True
```

- `Dict.clear()` removes all `key:value` pairs from the dictionary, does not return a value.

```
>>> Dict.clear()
>>> Dict
{}
```

- `Dict.copy()` returns a copy of `Dict`.

```
>>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
>>> Dict_copy = Dict.copy()
>>> Dict_copy
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
>>> Dict[2]='zzz'
>>> Dict
{1: 'a', 2: 'zzz', 3: 'c', 4: 'd', 5: 'e'}
>>> Dict_copy
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

- `Dict.fromkeys(S[,v])` - parameter `v` is optional - creates dictionary `Dict` with keys from `S` and values equal to `v`.

```
>>> Dict_fromkeys = {}
>>> S = 'abcd'
>>> Dict_fromkeys = Dict_fromkeys.fromkeys(S,10)
>>> Dict_fromkeys
{'a': 10, 'b': 10, 'c': 10, 'd': 10}
```

- `Dict.get(k[,d])` - parameter `d` is optional - returns `Dict[k]` if `k` is a key into the dictionary, otherwise returns `d`. If this method is called with only one argument – key, and such key does not exist in the dictionary, the value returned will be `None`.

```
>>> Dict_fromkeys
{'a': 10, 'b': 10, 'c': 10, 'd': 10}
```

```
>>> Dict_fromkeys.get('a')
10
>>> Dict_fromkeys.get('aa')
>>> Dict_fromkeys.get('aa',0)
0
```

- `Dict.pop`(`key`[,`error`]) - the `error` parameter is optional - it returns the value of the element corresponding to the key and removes it from the dictionary, if the key is not in the dictionary it returns the `error` value, and if the `error` value was not provided it throws `KeyError`.

```
>>> Dict
{1: 'a', 2: 'zzz', 3: 'c', 4: 'd', 5: 'e'}
>>> Dict.pop(1)
'a'
>>> Dict.pop(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> Dict.pop(1,"Key not found")
'Key not found'
```

- `Dict.popitem`() returns the pair (`key`, `value`) and removes it from the dictionary, and if the dictionary is empty it throws `KeyError`.

```
>>> Dict.popitem()
(5, 'e')
>>> Dict.popitem()
(4, 'd')
>>> Dict.popitem()
(3, 'c')
>>> Dict.popitem()
(2, 'zzz')
>>> Dict.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

- `Dict.setdefault`(`key`[,`error`]) - the `error` parameter is optional - returns the value of the element corresponding to the key, if the key is not in the dictionary, it returns the value `error` and inserts the pair (`key`:`value`) into the dictionary.

```
>>> Dict = {1:'a',2:'b',3:'c',4:'d',5:'e'}
>>> Dict.setdefault(1)
'a'
>>> Dict.setdefault(11)
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 11: None}
>>> Dict.setdefault(12,"Key not found")
'Key not found'
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e',
 11: None, 12: 'Key not found'}
```

- `Dict.update`([other]) does not return a value, updates the dictionary `Dict` with `key:value` pairs from the dictionary passed as parameter or another iterable consisting of (`key,value`) pairs.

```
>>> Dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 11: None,
 12: 'Key not found'}
>>> Dict1
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
>>> Dict.update(Dict1)
>>> Dict
{1: 'a', 2: 'xxx', 3: 'c', 4: 'd', 5: 'e', 11: None,
12: 'Key not found', 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
>>> Dict = {'x':2}
>>> Dict.update([('y',3),('z',4)])
>>> Dict
{'x': 2, 'y': 3, 'z': 4}
>>> Dict.update(name = 'Alice',surname = 'XXX')
>>> Dict
{'x': 2, 'y': 3, 'z': 4, 'name': 'Alice', 'surname': 'XXX'}
```

- `Dict.values`() returns a list of values from the dictionary.

```
>>> Dict.values()
dict_values([2, 3, 4, 'Alice', 'XXX'])
```

5.4.5. Named function arguments

At the end of the function argument list, you can optionally use the special form `**kwargs`. This is a dictionary (possibly empty) to which any additional named function arguments will be inserted in the form of `key:value` pairs, where the key is the argument name and the value is the value of that argument.

Listing 5.19: *Agrs passed using keywords*

```
def main():
  d = dict(jan = 31)
  print('1:',d)
  d = dict(jan = 31, feb = 28)
  print('2:',d)
  d = dict(jan = 31, feb = 28, mar = 31)
  print('3:',d)
  d = dict(jan = 31, feb = 28, mar = 31, apr = 30)
  print('4:',d)

if __name__=='__main__':
  main()
```

As a result of executing the program from listing 5.19, a dictionary is created each time from the arguments supplied using keywords, and information about their contents appears on the screen:

```
1: {'jan': 31}
2: {'jan': 31, 'feb': 28}
3: {'jan': 31, 'feb': 28, 'mar': 31}
4: {'jan': 31, 'feb': 28, 'mar': 31, 'apr': 30}
```

In listing 5.20, we show the use of the special form `**kwargs` in the function `show_sum`. This allows us to call this function with any number of named arguments.

Listing 5.20: *Any number of named args*

```
def main():
  show_sum(1,2,a=5,b=8)

def show_sum(x,y,**others):
  print("x:{0}, y:{1}".format(x,y), end='')
  print("\nValues in 'others':{}".format(list(others.values())))
  sum = x + y + sum(others.values())
  print( "The sum of the argument values is: {}".format(sum))

if __name__=='__main__':
  main()
```

As a result of executing the program from listing 5.20, we see on the screen the values of the positional arguments, the dictionary `other` and the calculated sum of all of them:

```
x: 1, y: 2
Values in 'other': [5, 8]
The sum of argument values is: 16
```

The general and complete form of the argument list in a function definition is shown in listing 5.21.

LISTING 5.21: *General form of function definition*

```python
def main():
  func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12)
  func(1, 2, 3, 4, 5, c = 5, d = 7, k1 = 11, k2 = 12)
  func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12, d = 7)
def func(a, b, *args, c, d = 6, **kwargs):
  # a and b are mandatory positional arguments
  print("a =", a, "b =", b )
  # args is a tuple of unnamed positional arguments
  print("args =", args)
  # c and d are arguments that are passed
  # only via keywords
  print("c =", c, "d =", d)
  # kwargs is a dictionary
  print("kwargs =", kwargs, "\n")
if __name__=='__main__':
  main()
```

As a result of executing the program from listing 5.21, the following information will be displayed on the screen:

```
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 6
kwargs = {'k1': 11, 'k2': 12}
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 7
kwargs = {'k1': 11, 'k2': 12}
a = 1 b = 2
args = (3, 4, 5)
c = 5 d = 7
kwargs = {'k1': 11, 'k2': 12}
```

5.5. Practice exercises

5.5.1. Lists

1. Define a function `F1(L)` that checks and returns how many even numbers there are in the list `L` passed as an argument to the function. Test the operation of the function `F1` in the function `main` for a list of 100 randomly generated integers from the range `⟨1,100⟩`.
2. Define a function `F2(L)` that checks and prints to the screen how many times each digit appears in the list `L` (for this purpose, use an auxiliary list whose indices represent digits and whose values represent the number of times they appear). Test the operation of the function `F2` in the function `main` for a list of 100 randomly generated integers from the range `⟨-50,50⟩`.
3. Define a function `F3(L)` that will calculate and return the arithmetic mean of the elements of the list `L.`. Test the `F3` function in the `main` function for a list of 100 randomly generated integers from the interval `⟨10,20⟩`.
4. Define a function `F4(L)` that will find the smallest element in the list `L` and return the last index under which this element is located and its value. Test the `F4` function in the `main` function for a list of 100 randomly generated integers from the interval `⟨-50,50⟩`.
5. Define a function `F5(L)` that checks and returns how many vowels are in the list `L.` Test the `F5` function in the `main` function for a list of 100 characters generated randomly from among the lowercase letters of the alphabet.
6. Define a function `F6(L)` that calculates and returns the difference between the largest and smallest element in the list `L` passed as an argument to the function. Test the `F6` function in the `main` function for a list of 1000 randomly generated integers from the interval `<1,100>`.
7. Define a function `F7(L, n)` that checks and returns `True` if the list `L` contains the nth power of 2; otherwise, the function returns `False`. Test in the `main` function the operation of the function `F7` for a list of 100 integers generated randomly from the interval `<1,64>` and n chosen randomly from the interval `⟨0,6⟩`.
8. Define the function `F8(L)`, which will create and return a list consisting of non-repeating elements of the list `L.` Test the operation of the function `F8` in the function `main` for a list of 100 integers generated randomly from the interval `⟨-50,50⟩`.
9. Write a program that creates a list consisting of lists that are permutations of the `5`-element set. Provide a random set of data for this set.
10. Create a list containing all the numbers that can be created from the digits `1`, `2`, `3` and `4`.

11. Define the function `F9(L, K)`, which will create and return a list consisting of common elements of the lists `L` and `K`. Test the function `main` operation of the `F9` function for two lists of 100 integers generated randomly from the interval `⟨-50,50⟩`.
12. Define a function `F10(L, K)` that will test and return `True` if the lists `L` and `K` have the same number and the same elements, `False` otherwise. Test the `F10` function in the `main` function for two lists in such a way that it covers all cases (when the lists are not the same length, when they have the same length and different elements, and when they have the same length and the same elements in different order).
13. For each of the subitems, we assume that we initially have an empty list `L = []`, which we fill with the appropriate numbers using one or two `for` iteration statements and display on the screen:
    (a) `1 2 3 4 5 6 7 8 9 10`
    (b) `0 2 4 6 8 10 12 14 16 18 20`
    (c) `1 4 9 16 25 36 49 64 81 100`
    (d) `0 0 0 0 0 0 0 0 0 0`
    (e) `0 1 0 1 0 1 0 1 0 1`
    (f) `0 1 2 3 4 0 1 2 3 4`
    (g) `1 -2 3 -4 5 -6 7 -8 9 -10`
    (h) `0 1 0 4 0 9 0 16 0 25 0 36 0 49 0 64 0 81 0 100`
14. Define a function `number_of_negatives` that returns as its result the number of negative elements of the list given as the only argument to this function. Test this function in the function `main`.
15. Define a function `product` that returns as its result the product of numbers that are elements of the list given as the only argument to this function. Test this function in the function `main`.
16. Define a function `minmax` that returns as its result a tuple of two numbers, the first of which is the minimum and the second of which is the maximum of the list given as the only argument to this function. For example, for a list:

    `a = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]`

    the function `minmax(a)` should return the tuple `(28,31)`. Test this function in the function `main`.
17. Write a function that calculates and returns the alternating sum of all the elements in the list. For example, for the list `[1, 4, 16, 9, 9, 7, 4, 9, 11]` the function should calculate and return as the result:

    `1 - 4 + 16 - 9 + 9 - 7 + 4 - 9 + 11 = 12`

    Test this function in the `main` function.
18. Write a program that reads the next number and adds it to the list if it is not already in the list. When the list contains ten numbers, the program prints the contents of the list and exits.

19. Given a list of employee salaries at a company, write a program that updates the list so that each employee receives a `10%` raise.
20. Write a program that implements the sieve of Eratosthenes that adds all the numbers from 2 to 10,000 to an initially empty list, then removes multiples of 2 (but not 2), multiples of 3 (but not 3), and so on, up to multiples of 100. The program then prints the numbers remaining in the list.
21. For each of the following items, write a function that performs the corresponding task for the list of integers that is its argument. Test each function in the `main` function.
    (a) Swaps the first and last elements in the list.
    (b) Moves all items one position to the right, and moves the last item to the beginning of the list, e.g. the list `[1 4 9 16 25]` is to be transformed into the list `[25 1 4 9 16]`.
    (c) Replaces all even items in the list with 0.
    (d) Replaces each item in the list, except the first and last, with the larger of its two adjacent items.
    (e) Removes the middle item if the list is odd in length, or the two middle items if the list is even in length.
    (f) Moves all even items in the list to the beginning of the list, preserving their order of appearance.
    (g) Returns the second largest item in the list.
    (h) Returns `True` if the list is sorted in ascending order. Otherwise, returns `False`.
    (i) Returns `True` if the list contains any two adjacent equal elements. Otherwise, returns `False`.
    (j) Returns `True` if the list contains any two equal elements (which do not need to be adjacent). Otherwise, returns `False`.
22. Given a list of 100 randomly generated integers from the interval `<-50,50>`, write a program that calculates the arithmetic mean, median, mode, and standard deviation of the elements in the list. What percentage of the list's elements are in the interval $< \overline{x} - \sigma, \overline{x} + \sigma >$, where $\overline{x}$ denotes the arithmetic mean and σ denotes the standard deviation?
23. Given lists `L1` and `L2` containing the digits of certain numbers from the ternary system, write a program that allows you to add these numbers (without using conversion to the decimal system).
24. Given two lists: `Lo`, `Lw`, write a program that, based on the grades in the `Lo` list and the corresponding weights from the `Lw` list, determines the grade, which is a weighted average of the data.

25. Define a function `PN` that returns a tuple of two numbers, the first of which is the number of even and the second - the number of odd elements in the list passed as the only argument to this function.
26. Write a function `how_many(L,a,b)` that returns information on how many natural numbers from the interval `(a,b)` are in the list `L`.
27. Given a list containing the terms of a finite sequence, write a program that checks whether the sequence is:
    (a) monotonic,
    (b) arithmetic.

5.5.2. Sets

1. Suppose we are given three sets: `set1`, `set2`, and `set3` of numbers generated randomly from the interval `<1,15>` of different lengths. Write instructions in a single program that:
    (a) Create a set of all elements that belong to `set1` or `set2`, but not both at the same time.
    (b) Create a set of all elements that belong to only one of the three sets `set1`, `set2`, and `set3`.
    (c) Create a set of all elements that belong to exactly two sets from `set1`, `set2`, and `set3`.
    (d) Create the set of all integers in the range 1 to 25 that are not in `set1`.
    (e) Create the set of all integers in the range 1 to 25 that are not in any of the three sets `set1`, `set2`, and `set3`.
    (f) Create the set of all integers in the range 1 to 25 that are not in any of the three sets `set1`, `set2`, and `set3`.
2. For each of the following, write a function that takes two strings as arguments and returns the following as result:
    (a) A set consisting of lowercase and uppercase letters that occur simultaneously in both strings.
    (b) A set consisting of lowercase and uppercase letters that appear in at least one string.
    (c) A set consisting of lowercase and uppercase letters that appear in no string.
    (d) A set consisting of all characters (other than letters) that appear in both strings at the same time.
    (e) A set consisting of all characters (other than letters) that appear in at least one string.
3. Given a string `n`, create a set containing the ASCII codes of the characters in the given string.

4. Implement the ancient Greek method of calculating prime numbers called the sieve of Eratosthenes. This method creates the set of all prime numbers no larger than n. Read the number into the variable `n`, then insert all numbers from `2` to `n` into the set `primes`. Then remove all multiples of `2` (except `2`), then all multiples of `3` (except `3`), and so on up to the square root of `n`. Finally, print the resulting set `primes`.
5. Modify the solution to the previous problem by creating a function `sieve(n)` that returns as its output the set of prime numbers no larger than `n`.
6. Create a list of ten unique names of people, and then randomly create three sets:
   - set A - a set of people who know English;
   - set N - a set of people who know German;
   - set F - a set of people who know French.

   Determine the following sets and determine the cardinality of each:
   - a set of people who know at least one language (from those given above);
   - a set of people who know exactly two languages;
   - a set of people who know three languages;
   - a set of people who know English but do not know German.
7. Given three integers, create the following sets:
   - a set with elements that are digits that appear in each of the three numbers;
   - a set of digits that do not appear in any of the given numbers.
8. Given a set `set1` containing letters and digits, create the sets `letters` and `digits` containing the letters and digits that appear in the set `set1`, respectively.
9. Given two lists containing one hundred randomly generated integers from the interval `<0,100>`, create the following sets:
   - a set A containing elements that appear in at least one of the lists;
   - a set B containing only elements that are common to both lists;
   - a set C containing elements that appear in both lists if they are squares of some natural numbers.

5.5.3. Dictionaries

1. Write a program that reads numbers provided by the user, adds them to a dictionary, and determines the number of their occurrences (the numbers are the keys, and the numbers of their occurrences are the values). If an element is provided that is not an integer, the program should print an appropriate message and continue. If the `<Enter>` key is pressed without first providing a value, the program should print the dictionary and exit.
2. For each of the following points, write a function that takes a string of characters as an argument and returns the following as the result:

(a) A dictionary containing information for each character in the string given as an argument how many times the character appears in the string.

(b) A dictionary containing information on how many times each letter appears in the string given as an argument if it appears at least once. For example, for the string `'abrakadabra'` the function should return the following dictionary: `{'b':2,'r':2,'d':1,'a':5,'k':1}` (the order of pairs in the dictionary is irrelevant).

(c) A dictionary like the one in the previous task, but created without distinguishing between lower and upper case letters.

(d) The letter that appears most often in the string given as an argument. If there are several such letters, the function can return any of them.

In the `main` function, test the given function by reading a string from the standard input and calling this function.

3. Write a program that reads successive lines from the standard input and splits the lines into words using the `split()` method. Then, for the received list of elements, it checks whether the next element is a number - if it is, it adds it to the dictionary. The program stops reading after reading an empty line. The program should print a dictionary in which the keys are numbers and the values are the numbers of their occurrences.
4. Modify the above program so that, for a given list of items, it checks whether a given item is a three-digit positive number and if so, adds it to the dictionary.
5. Write a function that returns the following for a string of characters passed as an argument:
   (a) a dictionary containing vowels and the number of occurrences;
   (b) a dictionary containing Polish diacritics and the number of occurrences;
   (c) a dictionary containing non-alphanumeric characters and the number of occurrences.

5.5.4. Various tasks

(a) Write a function `mediana(*numbers)` that returns the median of the numbers given as arguments as its result.

(b) Write a function `wypisz(*args)` that prints its arguments in alphanumeric order on subsequent lines.

(c) Write a function `kwsum(*numbers)` that returns the square of the sum of the arguments passed.

(d) Write a function `sumkw(*numbers)` that returns the sum of the squares of the numbers passed as arguments.

Chapter 6

Exception handling and working with files

In this chapter, we will show you how to handle exceptions in Python without interrupting your program. We will also describe the process of reading and writing data to a file using built-in functions that support this process.

6.1. Syntax errors

The most common types of errors are syntax errors, also known as "parsing" errors. On the standard output, the parser repeats the offending line and displays a small "arrow" pointing to the earliest point in the line where the error was found.

To illustrate a typical parsing error, consider the example of using a `while` loop with an intentional error.

```
>>> while True print("Hello")
File "<stdin>", line 1
while True print("Hello")
           ^
SyntaxError: invalid syntax
```

In the above example, the error was detected in the component preceding the arrow-symbol. The Python interpreter detected it on the `print` function call statement because of the missing colon character (`':'`) before the statement. The error message also displays the line number to let you know where to look for the error, and the name `<stdin>` - which is the standard input when working with the interpreter, or the file name - in case the input was a script.

6.2. Exception handling

Despite taking care of the syntactic correctness of an expression, it may happen that errors occur when trying to execute it. Errors detected during execution are called exceptions and, importantly, they do not necessarily have to lead to the termination of the program.

Most exceptions are not handled by programs, and their occurrence is reported by error messages. The following examples present the most common errors during program execution: attempting to divide by zero, referring to an undefined variable, or attempting to concatenate a string with a number.

```
>>> 10 * 1/0
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> 4 * 3 + spam
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>>
>>> "2" + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

The displayed string informing about the type of exception that has just occurred is its built-in name. In addition, the message describes in detail what happened, and the interpretation and meaning of this description depends on the type of exception that occurred.

The most important groups of errors that occur in Python include:

- `Syntax error` - invalid syntax, the interpreter terminates;
- `Type error` - an operation cannot be performed within the given type, e.g. adding a numeric value to a string;
- `Index error` - exceeding the scope of a list or tuple;
- `Value error` - unsupported value or data type;
- `Zero division error` - division by zero error.

If we know that an exception may occur and crash a program at a given point in the code, we should try to handle such a situation using a special block: `try...except...else` or `try...except...else...finally`.

First, according to their notation, the statements in the `try` clause (i.e. the statements between `try` and `except`) are executed. If no exception occurs, the `except` clause is omitted, and the execution of the `try` statement is considered to be complete. If an exception occurs during the execution of the `try` clause, the remaining, not yet executed statements are omitted. Then, depending on whether the exception type matches one of the exception types listed in the `except` clause(s), the code contained in the matched clause is executed, and the interpreter proceeds to execute the statements placed after the entire `try` statement. If an exception occurs that does not match any of the exceptions listed in the `except` clause(s), it is passed on to the next, external `try` statements. If no matching `except` clause is found there either, the exception is not caught: it becomes an unhandled exception, and program execution is halted with an appropriate message. To enable handling of multiple exceptions, a `try` statement can have more than one `except` clause, or a single `except` clause can contain the names of multiple exceptions, given as a list surrounded by round brackets, e.g.

```
except (TypeError, NameError):
  pass
```

The last one can be the `except` clause with the exception name(s) omitted, which will handle any exception. In the 6.1 listing on lines 15-18, we used this type of clause to print an error message and re-raise the caught exception with the `raise` statement, thus allowing the calling function, in the case of the program from the 6.1 listing - the `main` function, to catch the thrown exception.

We pay special attention to using the `except` construction with the exception name omitted, because its careless use can mask a real error occurring in the program!

The `try ... except` statement is also equipped with an optional `else` clause, which appears after all the specified `except` blocks. You can put code in it that will be executed if no exception is raised. Putting the code in the `else` clause instead of outside the entire `try` clause is safer because you can avoid accidentally throwing an exception that was not previously raised by the code protected in the `try` clause.

LISTING 6.1: *Handling zero division error and any other exception*

```
1 def demo1(x):
2   try:
3     print(1/x)
4   except ZeroDivisionError:
5     print("x =", x, ":You cannot divide by zero")
6   except:
7     print("x =", x, ":unexpected exception")
8     raise
```

```
9   else:
10    print("You did it :-)")
11
12 if __name__ == "__main__":
13   a = float(input("Enter a number: "))
14   demo1(a)
15   try:
16     demo1(str(a))
17   except:
18     print("An exception ocurred again")
```

In listing 6.1, we show handling of the division by zero exception (line 4) with consideration of other exceptional situations (line 6). Possible calls causing exceptions may look like this:

- after the user provides the value `0` or `0.0`:

```
Enter a number: 0
x = 0.0 : You cannot divide by zero
Executing <demo> function again
For arg  <str>
x = 0.0 : unexpected exception
An exception ocurred again
```

- after the user enters a non-zero value:

```
Enter a number: 12.0
0.08333333333333333
Udało się :-)
Executing <demo> function again
For arg  <str>
x = 12.0 : unexpected exception
An exception ocurred again
```

Each time the `demo1` function is called again in the `try` block in the `main` function (lines 15-18), the code contained in the `demo1` block is executed (lines 6-8) in the body of the `demo1` function and the exception, which is already handled in the calling function, is raised again.

Hence, as a result of calling the program from listing 6.1 in both of the above-mentioned examples, the last two lines of the message on the screen have similar content.

In the `try` block, the last clause may optionally be `finally`, the code of which will always be run at the end of exception handling. In listing 6.2, we present the use of this clause, which will always print information about the end of the exception handling block to the screen at the end of exception handling (lines 9-10). In the `except` clause, you can use the phrase `as` and a variable name, e.g. `er` as we did on line 4. Then the exception object is assigned to the `er` variable, thanks to which we can obtain additional information about the exceptional situation that occurred and, for example, display it on the screen (line 6).

Listing 6.2: *Using* `finally` *clause*

```
1  def demo2(x):
2    try:
3      print(1/x)
4    except ZeroDivisionError as er:
5      print("x =", x, ":You cannot divide by zero")
6      print("Exception",type(er).__name__,"(",er,") occurred")
7    else:
8      print("No exceptions")
9    finally:
10     print("Always at the end of the try!")
11
12 if __name__ == "__main__":
13   a = float(input("Enter a number: "))
14   demo2(a)
```

In listing 6.3, we present on line 4, the throwing of an exception by a function which is an object of class `ValueError` together with an error message, which in this case is a string argument of the object being created. In the function `main` in the clause `except`, which handles this type of exception, the variable `e` is associated with it, thanks to which a previously formulated error message will be printed to the screen.

Listing 6.3: *Throwing an exception*

```
1  def demo3():
2    x = int(input("Enter an integer number:   "))
3    if x < 0: raise ValueError("The number *cannot* be negative")
4    else: return x
5
6  if __name__=='__main__':
7    try:
8      print(demo3())
9    except  ValueError as e:
10     print(e)
```

In listings 6.4 and 6.5, we defined two functions to convert strings to values of type `int` and `float` respectively, handling exceptional situations related to it.

Listing 6.4: *Support for conversion to* `int` *type*

```
def inputInt(prompt):
  while True:
    try:
      a = int(input(prompt))
    except:
      print("This is not an integer")
      print("Retry until you get it!")
    else:
      return a

if __name__=='__main__':
  print(inputInt("Enter an integer:"))
```

Listing 6.5: *Support for conversion to* `float` *type*

```
def inputFloat(prompt):
  while True:
    try:
      a = float(input(prompt))
    except:
      print("This is not a float")
      print("Retry until you get it!")
    else:
      return a

if __name__=='__main__':
  print(inputFloat("Enter a float:"))
```

As we mentioned earlier, the `raise` clause is used to report an error on its own if a situation occurs in the code, the existence of which we want to inform the user - more of another programmer who will use our code than the end user (line 2 of listing 6.6). The `inst` variable from line 3 of listing 6.6 is associated with the exception instance `Exception`, which confirms the type of the variable (line 4). This instance has an attribute `args` that stores arguments in a shorthand (line 5). Built-in exception types also define a method `__str__()`, which prints all arguments without the need to refer to the `args` attribute (line 6). We strongly recommend against handling exceptions by handling the base `Exception`. In an exceptional situation, you should match the appropriate built-in exception type as closely as possible or define your own class to handle it.

Listing 6.6: *Throwing* `Exception`

```
try:
  raise Exception('one', 'two')
except Exception as inst:
  print(type(inst))
  print(inst.args)
  print(inst)
  x, y = inst.args # Unpacking tuple inst.args
  print('x =', x)
  print('y =', y)
```

In turn, listing 6.7 presents handling of an exceptional situation related to the occurrence of an error on line 3 when trying to convert a number from hexadecimal to decimal, where in the number `"1OO"` the capital letter `'O'` was used instead of zero.

Listing 6.7: *Hexadecimal to decimal conversion support*

```
1 def main():
2   print(str_to_int("ABCD"))
3   print(str_to_int("1OO"))
4 
5 def str_to_int(string):
6   try:
7     return int(string, 16)
8   except ValueError as e:
9     print("Value error:", e)
10 
11 if __name__ == "__main__":
12   main()
```

In listing 6.8, we will use exception handling and command line arguments to calculate the smallest sum of numbers `1 + 1/2 + 1/3 ...` greater than the value of the passed argument, which should be a real number.

Listing 6.8: *The sum of the reciprocals of natural numbers*

```
import sys

def main():
  if len(sys.argv) != 2: a = 2.0
  else:
    try: a = float(sys.argv[1])
    except: a = 3.0
  print(sum_of_reciprocals(a))
```

```
9 def sum_of_reciprocals(a):
10   n, s = 0, 0.0
11   while s <= a:
12     n += 1
13     s += 1 / n
14   return s
15
16 if __name__ == "__main__":
17   main()
```

In the function `main` on line 5, the condition of correctness of the program call from listing 6.8 is tested. If the program was called for zero or more than one argument, then the value `2.0` is assumed as the limit for the calculated sum of reciprocals of subsequent natural numbers. Otherwise, in the `try` block (line 6), we read and convert the value of the argument `sys.argv[1]` to the `float` type. If it is not a number, the conversion will fail, and an exception will be raised, which will then be handled in the `except` block (line 7), and the value `3.0` will be assumed as the limit. Thanks to these procedures, the call to the function `sum_of_reciprocals` will always return and print to the screen the result of the calculations performed in its body.

```
~$ python3 main_arg_4.py 6.0
6.004366708345567
~$ python3 main_arg_4.py
2.083333333333333
~$ python3 main_arg_4.py 6.0 4
2.083333333333333
~$ python3 main_arg_4.py "Alice"
3.0198773448773446
```

Listing 6.9 shows an example of a program that handles an exception with the full `try-except-else-finally` statement, using the `raise` statement (line 6) and an error class defined by the program creator (line 1).

LISTING 6.9: *Throwing an arror using custom error class*

```
1 class NonPositiveError(Exception): pass
2
3 try:
4   a = float(input('Enter length of a square side:'))
5   if a <= 0:
6     raise NonPositiveError
7 except ValueError:
8   print('Incorrect format')
```

```
except NonPositiveError:
  print('The number must be positive')
else:
  print('Area of the square', a**2)
finally:
  print('Program end.')
```

6.3. Working with files

From the operating system's perspective, a file is a sequence of bytes of a specified length, stored on a permanent medium. In Python, a distinction is made between text files and binary files. A text file is a sequence of characters encoded by default in the UTF-8 encoding, while a binary file is a sequence of bytes. UTF-8 is a Unicode encoding system that uses from 1 to 4 bytes to encode a single character, fully compatible with ASCII. The compatibility of the UTF-8 encoding with ASCII means that each string of ASCII characters is also a string of characters in UTF-8. In Python, files are also called streams. Access to files is obtained in Python through the standard (available without importing additional modules) file class. Creating an object of this class involves opening the file, allowing data to be written and/or read. The open function asks the operating system to allow access (read/write) to the file. When this is successful, an object of the file class is returned, which has methods and attributes that allow certain operations to be performed on the file.

The most commonly used form of calling the open function is the one with two arguments: open(file_name, mode). The first argument is a string specifying the file name. The second argument is a string describing the file opening mode. The mode argument is optional: if it is missing, the file will be opened in read-only text format.

The important attributes of a file object are: name (file name), mode (file opening mode), encoding (encoding only exists for text files), and closed ("Is it closed?" Returns True if the file is closed otherwise False). Calling the close method on a file object closes the file and immediately releases all system resources used by the file.

```
>>> f = open('/etc/passwd')
>>> f.name
'/etc/passwd'
>>> f.mode
'r'
>>> f.encoding
'UTF-8'
```

```
>>> f.closed
False
>>> f.close()
>>> f.closed
True
```

6.3.1. File opening modes

`Read-only file opening modes`:

- In each case from table 6.1, after opening a file, the file pointer is placed at its beginning.
- If the file with the given name does not exist, an exception is thrown: `FileNotFoundError`.

Table 6.1: File opening modes for reading

`Mode`	`Description`
`r`	Opens a file in text format for reading only (this is the default mode).
`rb`	Opens a file in binary format for reading only.
`r+`	Opens a file in text format for reading and writing.
`rb+`	Opens a file in binary format for reading and writing.

`Write file opening modes`:

- In each case from table 6.2, if a file with the given name exists, its contents are overwritten.
- If a file with the given name does not exist, a new file is created.

Table 6.2: File opening modes for writing

`Mode`	`Description`
`w`	Opens a file in text format for writing only.
`wb`	Opens a file in binary format for writing only.
`w+`	Opens a text file for reading and writing.
`wb+`	Opens a binary file for reading and writing.

`File opening modes for appending`:

- In each of the cases from table 6.3, after opening the file, the file pointer is placed at the end of it.
- If the file with the given name does not exist, a new file is created.
- In turn, in each of the cases from table 6.4, if the file with the given name exists, an exception is thrown: `FileExistsError`.

Table 6.3: File opening modes for appending

Mode	Description
`a`	Opens a file in text format for appending only.
`ab`	Opens a file in binary format for appending only.
`a+`	Opens a file in text format for reading and appending.
`ab+`	Opens a file in binary format for reading and appending.

Table 6.4: File opening modes for appending

Mode	Description
`x`	Opens a file in text format for writing only.
`xb`	Opens a file in binary format for writing only.
`x+`	Opens a file in text format for reading and writing.
`xb+`	Opens a file in binary format for reading and writing.

Files are typically opened in text mode (*text mode*), which means that we read and write strings to and from the file, which are encoded according to the optional third argument `encoding`. The default encoding is platform-dependent. Since `UTF-8` is the modern standard encoding, it is recommended to use `encoding="utf-8"` unless you know you need to use something else. In binary mode (*binary mode*), data is read and written as `bytes` objects. You cannot specify an encoding for these types of files when opening them.

6.3.2. Selected methods for text file objects

We assume that a file object `f` has already been created and associated with a physical file on disk called `file1.txt` with the following contents:

```
First line.
Second line.
Third line.
The last, next line is empty.
```

We will not care about the file opening mode here, we will only illustrate how to use the methods of the `f` object.

- `f.readable()` – returns `True` if the file can be read, and `False` otherwise, e.g.

  ```
  >>> f.readable()
  True
  ```

- `f.writable()` – returns `True` if the file is writable, and `False` otherwise, e.g.

  ```
  >>> f.writable()
  False
  ```

- `f.close()` -closes the file and releases all system resources associated with opening and handling the file.

  After `f.close()` is called, any attempt to use a method of the `f` file object will generate an error:

  `ValueError: I/O operation on closed file`.

  If the file is not closed by the program, the `close` method will be called by the interpreter when the object is finally destroyed. For example:

  ```
  >>> f.close()
  >>> f.readable()
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: I/O operation on closed file
  ```

- `f.read(size)` - reads the contents of a file by reading some amount of data (`size`) and returning it as a string (in text mode) or a byte object (in binary mode). `size` is an optional numeric argument, and if omitted or negative, the entire contents of the file are read and returned. Otherwise, at most `size` characters (in text mode) or `size` bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` returns an empty string (`''`). For example:

  ```
  >>> f.read()
  'First line.\nSecond line.\nThird line.\n
  The last, next line is empty.\n\n'
  ...
  >>> f.read(12)
  'First knows'
  ...
  >>> f.read(-1)
  'First line.\nSecond line.\nThird line.\n
  The last, next line is empty.\n\n'
  ```

- `f.readline()` - returns a string whose content is a single line read from the file. The newline character (`\n`) is left at the end of the string and is only omitted from the last line of the file if the file does not end with a newline. After reading all lines of the file, the next call returns an empty string (`''`). An empty line is represented by `'\n'`. For example:

```
>>> f.readline()
'First line.\n'
>>> f.readline()
'Second line.\n'
>>> f.readline()
'Third line.\n'
>>> f.readline()
'The last, next line is empty.\n'
>>> f.readline()
'\n'
>>> f.readline()
''
```

- `f.readlines()` - returns a list of strings that are the successive lines of the file, from first to last.

  Another way to read lines from a file is to loop over the file object, which is memory efficient and fast, e.g.

```
>>> for line in f: print(line, end='')
...
First line.
Second line.
Third line.
The last, next line is empty.

>>>
```

  Another way to read all the lines from a file is to use the `list` function to create a list of them, e.g.

```
>>> list(f)
['First line.\n', 'Second line.\n', 'Third line.\n',
'The last, next line is empty.\n', '\n']
```

- `f.write(string)` – writes the contents of `string` to a file. Returns the number of characters written to the file. For example:

```
>>> f.write('This is another line.\n')
24
```

  Non-string types must be converted to a string (in text mode) or a byte object (in binary mode), respectively, before being written to the file, e.g.:

```
>>> number = 123
>>> f.write('This is an integer' + str(number) + '\n')
28
```

- `f.writelines(lines)` – writes a list of `lines` strings to a file. Line separators are not added, so remember to manually add the separator at the end of each line. For example:

  ```
  >>> subtitles = ['First\n', 'Second\n','Third\n']
  >>> f.writelines(napisy)
  ```

- `f.flush()` - writes data from a buffer to a file without closing it. NOTE: on some file objects it may be an empty operation.

6.3.3. Examples of working with text files

In listing 6.10, we present a program in which we create a file with character encoding `encoding = 'UTF-16'` (lines 1-3), and then try to read from a file opened with the default value of the argument `encoding = 'UTF-8'` (lines 4-6). The encoding mismatch causes an exception to be thrown:

```
UnicodeDecodeError: 'utf-8' codec can't decode
byte 0xff in position 0: invalid start byte
```

Listing 6.10: *Character encoding in files*

```
1 f = open("utf16.txt", "w", encoding="UTF-16")
2 f.write("Programming is fun!")
3 f.close()
4 f = open("utf16.txt")
5 content = f.read()
6 f.close()
```

In the next example shown in listing 6.11, we create the variable `rhyme`, which is a multi-line string (lines 1-6). Immediately after opening this string, the continuation character '\' is used. If it were not there, the string would contain an additional empty line at the very beginning. We open a file named `'rhyme.txt'` for writing (line 7) and write the text divided into lines to it (line 8). It should be remembered that data is not written to disk immediately but only when a larger amount of it has been collected. Therefore, if we want to continue working on the file but in read mode, we should perform the operation opposite to opening, i.e. `close` (line 9). Then, the data will be written to disk, and the file object will be deleted. However, if we want to start reading this file without closing it first - we need to make sure that our changes are really on the disk, and here, the `flush` method comes in handy (comment on line 9). After updating the file `'rhyme.txt'`, it is reopened, but this time for reading (line 10), and its contents are read line by line and printed to the screen (lines 11 and 12).

Listing 6.11: *Example of repeatedly opening a file*

```
rhyme = '''\
Eeny, meeny, miny, moe,
Catch a tiger by the toe.
If he hollers, let him go,
Eeny, meeny, miny, moe.
'''
f1 = open('rhyme.txt', 'w')
f1.write(rhyme)
f1.close() #; f1.flush()
f2 = open('rhyme.txt')
for wers in f2:
  print(wers)
```

Closing files is often omitted (as is the case at the end of the code in the example from listing 6.11) because we know that Python will close the file anyway the program completion will close everything for us and release resources associated with the file object. To prevent errors resulting from not closing a file when required, Python introduced the `with` instruction and equipped opened objects (such as `file`) with the functionality to automatically close them because the object is automatically closed when the `with` block ends. Listing 6.12 illustrates the correct use of this instruction.

Listing 6.12: *Using* `with`

```
with open('rhyme.txt') as file:
  for lines in file:
    print(line, end='')
```

Listing 6.13, shows the code of a program that reads successive lines from a text file using the `while` instruction (lines 3-7) and the `readline` method (line 4). Reading the last line from the file, which is empty (lines 5 and 6), causes exiting the loop and terminating the program.

Listing 6.13: *Using* `while` *loop to read file*

```
1 def main():
2   f = open("rhyme.txt", "r")
3   while True:
4     line = f.readline()
5     if len(line) == 0:
6       break
7     print(line, end = "")
8 if __name__ == '__main__':
9   main()
```

In listing 6.14, we present the code of a program that creates a sorted copy of a text file by writing all lines at once. By reading the entire contents of the file as a list of its lines (line 3), we can use the method for sorting the list in ascending order (line 5), which we then write to the output file (line 7).

LISTING 6.14: *Creating a row-sorted copy of a file*

```
1  def main():
2    f = open("rhyme.txt", "r")
3    lines = f.readlines()
4    f.close()
5    lines.sort()
6    g = open("posortowanyrhyme.txt", "w")
7    g.writelines(lines)
8    g.close()
9
10 if __name__ == '__main__':
11   main()
```

In the program in listing 6.15, the user specifies a file name (line 2). If a name is not specified, the program prints a message to the screen and exits (lines 3 and 4). Otherwise, it tries to open the file for reading in the `try` block (line 6), and if that fails, the program exits after printing a message to the screen (line 9). If the file is successfully opened, the code in the `else` clause is executed (lines 11-16), which creates a dictionary of characters and the number of their occurrences in the file and then prints it to the screen.

LISTING 6.15: *Returns a dictionary of characters and the number of their occurrences in the file*

```
1  def main():
2    name = input("Enter file name: ")
3    if len(name) == 0:
4      print('No file name entered!')
5    else:
6      try:
7        f1 = open(name, "r")
8      except:
9        print("Failed to open file")
10     else:
11       D = {}
12       for lines in f1:
13         for character in line:
14           D[char.lower()] = D.get(char.lower(),0) + 1
15       f1.close()
```

```
        print(D)
if __name__ == '__main__':
    main()
```

We suggest using exception handling when working with files, where, especially in the case of an attempt to open a file, this operation may end in failure. It is worth informing the user why the program was terminated or handling the situation appropriately.

6.4. Binary files

Before we move on to presenting operations on binary files, let us discuss binary sequential types - the `bytes` and `bytearray` classes.

Objects of class `bytes` are immutable sequences of singletons bytes. Each byte has an integer value between 0 and 255. Most binary protocols are based on encoding `ASCII` text string, so the `bytes` class offers methods that only work with `ASCII` compatible data. Only the characters `ASCII` are allowed in byte literals (regardless of the declared source code encoding).

The `bytes([source[,encoding[,errors]]])` function allows you to create an object of class `bytes`, where:

- `source`: If `source` is a string (`str`), `encoding` (and optionally `errors`) is required; then the string is converted to bytes using `str.encode()`. If `source` is an integer (`int`), the sequence will be that size and initialized with zero bytes. If `source` is an iterable, then it must be an integer object in the range 0 to 255, which are used as the initial contents of the sequence. Without an argument, `bytes()` creates a sequence of size 0.
- `encoding`: optional parameter. Required if `source` is a string. Example: `'utf-8'`, `ascii` etc.
- `errors`: optional parameter for `source` which is a string. Depending on your needs, it can have values such as: `'strict'` - throw exceptions, `'replace'` - replace with `'?'`, `'ignore'` - skip the invalid character, etc.

Creating an empty object of class `bytes`:

```
>>> x = bytes()
>>> print(x)
b' '
```

Creating a 3-byte sequence filled with zero bytes:

```
>>> x = bytes(3)
>>> print(x)
b'\x00\x00\x00'
```

Creating a sequence of length list filled with its values:

```
>>> x = bytes([1,2,3])
>>> print(x)
b'\x01\x02\x03'
```

Creating a character sequence in the code `'UTF-8'`:

```
>>> x = bytes('Python','utf-8')
>>> print(x)
b'Python'
```

Creating a character sequence in the `'ASCII'` code:

```
>>> x = bytes('Python', 'ascii')
>>> print(x)
b'Python'
```

Attempting to create a `bytes` object for a string source without specifying an encoding fails with the error `TypeError`:

```
>>> bytes('foal')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding
```

Specifying an encoding where characters outside of this encoding are found in the source also results in the interpreter reporting an error, but this time an encoding error. `UnicodeEncodeError`:

```
>>> bytes('źrebak','ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can‘t encode character
'\u017a' in position 0: ordinal not in range(128)
```

A similar message is obtained when we use the third argument of the function `bytes` with the values `'strict'`:

```
>>> bytes('źrebak','ascii','strict')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can‘t encode character
'\u017a' in position 0: ordinal not in range(128)
```

Using the 'ignore' option as the third argument to a `bytes` object will ignore any character that does not belong to the specified code set and create an object from the remaining valid characters:

```
>>> bytes('foal','ascii','ignore')
b'rebak'
```

If we use the 'replace' option, each character outside the code set will be replaced in the created `bytes` object with the question mark '?':

```
>>> bytes('foal','ascii','replace')
b'?rebak'
```

Finally, the correct creation of the `bytes` object taking into account all the characters present in the source:

```
>>> bytes('foal','utf-8')
b'\xc5\xbarebak'
```

We will also not create a `bytes` object if the values of the numeric range specified as the source are outside the allowed values:

```
>>> bytes(range(100,300))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
```

Also when the source is a list of values that is not in the required set:

```
>>> bytes([1,2,300])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
```

And once again creating a `bytes` object consisting of Polish diacritics:

```
>>> bytes("ąćęłńó", "utf-8")
b'\xc4\x85\xc4\x87\xc4\x99\xc5\x82\xc5\x84\xc3\xb3'
```

which is then converted into a list of code numbers of these characters:

```
>>> list(bytes("ąćęłńó", "utf-8"))
[196, 133, 196, 135, 196, 153, 197, 130, 195, 179]
```

Listing 6.16 presents the code of a program that transforms strings into byte sequences. First, we create two variables (lines 1-2): `a`, which is a string, and `c`, which is a string of bytes. Then, we encode the string `a` to type `bytes` (line 3) and check whether the object d is of type `bytes` (lines 4-7). By comparing the object `d` with the reference object `c`, we learn whether the encoding was successful (lines 8-11).

Listing 6.16: *Transforming* `str` *to* `bytes`

```
1  a = 'Python Programming Basics'  # <class 'str'>
2  c = b'Python Programming Basics' # <class 'bytes'>
3  d = a.encode('ASCII')
4  if isinstance(d,bytes):
5    print(type(d))
6  else:
7    print(type(a))
8  if (d==c):
9    print ("Encoding completed successfully")
10 else:
11   print("Encoding failed")
```

Listing 6.16 presents the code of a program that converts byte sequences into strings. First, we create two variables (lines 1-2): `a`, which is a string, and `c`, which is a string of bytes. Then, we decode the byte sequence `c` to type `str` (line 3) and check whether the object d is of type `str` (lines 4-7). By comparing the object `d` with the reference object `a`, we learn whether the encoding was successful (lines 8-11).

Listing 6.17: *Transforming* `bytes` *to* `str`

```
1  a = 'Python Programming Basics'  # <class 'str'>
2  c = b'Python Programming Basics' # <class 'bytes'>
3  d = c.decode('ASCII')
4  if isinstance(d,str):
5    print(type(d))
6  else:
7    print(type(a))
8  if (d==a):
9    print ("Encoding completed successfully")
10 else:
11   print("Encoding failed")
```

Objects of class `bytearray` are mutable equivalents objects of class `bytes`. Function `bytearray([source[,encoding[,errors]]])` allows you to create an object of class `bytearray` in a similar way to the `bytes` function:

```
bytearray()
bytearray(int)
bytearray(iterable_of_ints)
bytearray(string, encoding[, errors])
bytearray(bytes_or_buffer)
```

Listings 6.18 and 6.19 present certain operations that can be performed on both `bytes` and `bytearray` type objects, i.e. referencing elements of both objects by writing them to the screen (lines 3 and 4), writing fragments of these objects defined by ranges (lines 7-9). The difference is that an attempt to change an element of the `bytes` object (6.18: line 11) will result in a type error. Meanwhile, in listing 6.19, on lines 9-16, we change the case of the letters that make up the `bytearray` sequence, resulting in the text `bytearray(b'pYTHON')` on the screen.

Listing 6.18: *Operations on object* `bytes`

```
1  r = b"Python" #<class 'bytes'>
2
3  for and in r:
4    print(chr(i), end=' ') # P y t h o n
5
6  print()
7  print(r[0:2]) # b'Py'
8  print(r[0])   # 80
9  print(r[:-2]) # b'Pyth'
10
11 #r[0] = 's'
12 '''
13 Traceback (most recent call last):
14   File "w10-p7.py", line 8, in <module>
15   r[0] = 's'
16 TypeError: 'bytes' object does not support item assignment
17 '''
```

Listing 6.19: *Operations on object* `bytearray`

```
1  t = bytearray(b"Python")     # <class 'bytearray'>
2
3  for i in t:
4    print(chr(i), end=' ' )    # P y t h o n
5  print()
6  print(t[0:2])                # bytearray(b'Py')
7  print(t[0])                  # 80
8  print(t[:-2])                # bytearray(b'Pyth')
```

```
i = 0
while i < len(t):
  if(chr(t[i]).isupper()):
    t[i] = ord(chr(t[i]).lower())
  else:
    t[i] = ord(chr(t[i]).upper())
  i += 1
print(t) # bytearray(b'pYTHON')
```

In Python 3.x, the `bytes` type contains sequences of 8-bit values, while the type `str` contains sequences of `Unicode` characters. For this reason, the two methods we propose may be useful:

- method `to_str`, which takes data of type `str` or `bytes` and always returns data of type `str`;

```
def to_str(bytes_or_str):
  """ to_str """
  if isinstance(bytes_or_str, bytes):
    value = bytes_or_str.decode("utf-8")
  else:
    value = bytes_or_str
  return value # Always str.
```

- method `to_bytes`, which takes data of type `str` or `bytes` and always returns data of type `bytes`.

```
def to_bytes(bytes_or_str):
  """ to_bytes """
  if isinstance(bytes_or_str, str):
    value = bytes_or_str.encode("utf-8")
  else:
    value = bytes_or_str
  return value # Always bytes.
```

6.4.1. Selected methods of binary file objects

- `f.readlines()` – for a stream opened in binary mode reads all lines from a file and returns a list of objects of class `bytes`.
- `f.readline()` – for a stream opened in binary mode reads one line and returns an object of class `bytes`.
- `f.read(size=-1)` – for a stream opened in binary mode reads at most `size` bytes from stream (for `size > 0`) and returns an object of class `bytes`. For `size < 0` it reads all bytes from the current position to the end of the file.

- `f.write(buffer)` – for a stream opened in binary mode writes the contents of buffer `buffer` to a file. Returns the number of bytes written, which is always the length of the buffer in bytes.
- `f.tell()` – returns the current position of the stream.
- `f.seek(offset, whence=SEEK_SET)` – sets the stream position. Possible values for the second argument are:
  - `SEEK_SET` or 0 – sets the position relative to the beginning of the stream;
  - `SEEK_CUR` or 1 – sets the position relative to the current position;
  - `SEEK_END` or 2 – sets the position relative to the end of the stream.

  All these values are defined in the `io` module. So after an import statement of the form `import io`, they should be referenced by their fully qualified name, e.g. `io.SEEK_SET`.

To illustrate operations on binary files, in listing 6.20, we present a program for copying a graphic file in binary mode. To test the program, you must first prepare and place a graphic file in the directory where the Python file is located a graphic file - in our case,s it was a file named `image.png`. We open this file for reading in binary mode (line 1) and then open the second file, which is to be its copy, this time for writing in binary mode (line 2). We read cyclically `1kB` of data from the input file (line 6) until the end of the file is read - this will cause exiting the loop (lines 7 and 8) and writing each of them to the output file (line 9). Finally, we close both files (lines 10 and 11).

Listing 6.20: *Copying file in binary mode*

```
1  def main():
2    f = open("obraz.png", "rb")
3    g = open("kopia.png", "wb")
4
5    while True:
6      buf = f.read(1024)
7      if len(buf) == 0:
8        break
9      result = g.write(buf)
10   f.close()
11   g.close()
12
13 if __name__ == "__main__":
14   main()
```

Listing 6.21 presents a program that uses program call arguments, which contain the name of a file provided by the caller, the size of which in bytes is to be calculated. If the user provides an incorrect number of arguments, information about the correct form of the program call is sent to the screen, and the program itself terminates using the `exit` function from the `sys` module for this purpose. If the file is identified on the disk (line 7), the `f_size` function is called, which opens the file with the name provided as an argument for reading in binary mode and moves the file pointer to its end (line 15), the current position of the stream is read (line 16), which is simultaneously returned after closing the file (line 17), as its size (line 18).

LISTING 6.21: *Using the* `tell` *and* `seek` *methods*

```
1  import io, sys, os
2
3  def main():
4    if len(sys.argv) != 2:
5      print("Using the program: python", sys.argv[0], "plik")
6      sys.exit(1)
7    if os.path.isfile(sys.argv[1]):
8      print("File size", sys.argv[1] + ":", end='')
9      print(f_size(sys.argv[1]))
10   else:
11     print("File: ", sys.argv[1], " not found")
12
13 def f_size(filepath):
14   f = open(filepath, "rb")
15   f.seek(0, io.SEEK_END)
16   size = f.tell()
17   f.close()
18   return size
19
20 if __name__ == "__main__":
21   main()
```

6.5. Practice exercises

1. Write a program that performs the following tasks:
   - Opens a file named `hello.txt`.
   - Writes the message `"Hello, World!"` to this file.
   - Closes this file.
   - Opens the same file again.
   - Reads a message from a file into a string variable and prints it.
   - Closes this file.

2. Write a program that:
   - opens a file with the name provided by the user, then saves the information provided by the user, and then closes the file;
   - opens the above file, reads and writes its contents to the screen, and then closes it.
3. Write a program that reads a text file line by line and writes the read lines to an output file preceded by a line number. The user should be asked for the file names at the beginning of the program.
4. For each of the following points, write a function that takes a filename string as arguments, tries to open the file for reading, and then, using the `readline`, reads the following lines from this file. After reading all the lines, the function closes the file. The function returns the following as its result:
   (a) length of the longest line in this file;
   (b) the longest line from this file (if there are more lines of the same length, the function returns the first of those lines);
   (c) a tuple whose first element is the length of the longest line in this file, and whose second element is that line.

   In case the file failed to open, each function should return `None`. Test the above functions in the `main` function.
5. Write a program `catfiles.py` that combines the contents of several files into one file. For example, `python3 catfiles.py r1.txt r2.txt r3.txt book.txt` creates an output file `book.txt`, which contains the contents of the files in the order `r1.txt`, `r2.txt`, and `r3.txt`. The target file is always the last file specified on the command line.
6. Write a program `reverse.py` that reads all the lines from a file and writes them in reverse order to the output file. For example, if the file `input.txt` consists of the lines:

   ```
   Mary had a little lamb
   Its fleece was white as snow
   And everywhere that Mary went
   The lamb was sure to go.
   ```

   this after calling `~$ python3 reverse.py input.txt output.txt` the output file `output.txt` will contain the lines:

   ```
   The lamb was sure to go.
   And everywhere that Mary went
   Its fleece was white as
   Mary had a little lamb
   ```

7. Write a function `reversed_line` that takes a string as its argument and returns a reversed string, where if the last character of the string is a newline (`'\n'`), it leaves it in place. Test this function in `main`.
8. Write a program `reverse_lines.py` that replaces each line in a file with its reverse. For example, if we run the program via
   `python reverse_lines.py hello.py`
   this is the content of the file `hello.py`

```
# My first Python program.
print("Hello, World!")
```

   will change to

```
.margorp nohtyP tsrif yM #
)"!dlroW ,olleH"(tnirp
```

   After running `reverse_lines.py` again on the same file, we will recover the original file. In your program, use the function from the previous task called `reversed_line`.
9. It is known that the first 8 bytes of a file in `PNG` format have the following values:

   `137, 80, 78, 71, 13, 10, 26, 10`

   Write a program `is_png.py` that checks whether the file supplied as its argument is an image in `PNG` format.
10. Generalize the program from task 9 so that it can be given on the call line program more files.
11. Write a two-argument function `encrypt` whose first argument `buf` is an object of type `bytes` and whose second argument `mask` is an integer from the range `range(256)`. This function returns an object of class `bytes`, each element of which has a value that is the result of applying the disjunctive operator to the corresponding element of `buf` and the number `mask`. In the function `main`, test the correctness of the `encrypt` function. For example, the call `encrypt(b'\x00\01\x02', 255)` will return an object `b'\xff\xfe\xfd'`. In turn, calling `encrypt(b'Python', 255)` will return object `b'\xaf\x86\x8b\x97\x90\x91'`.
12. Write a program `cykl_xor.py` consisting of the function `main` and the function `encrypt` from the previous task. The `main` function checks whether the program was called with three arguments. If not, it prints an appropriate message and terminates the program. Then, it tries to open in binary mode to read the file with the name given as the first argument of the program, and then it tries to open the file with the second name in binary mode for writing the program argument. In case of failure, it prints an appropriate message and exits program execution. Furthermore, if the third argument is not a number in the interval `range(256)`, then it prints an appropriate message and terminates the program. Otherwise, the `main` function

reads successive 32 byte chunks of data from the input file in a loop, calls the `encrypt` function for the next chunk and the number given as the third argument to the program, and then writes the result returned by the `encrypt` function to the output file.

13. Write a two-argument function `caesar_cipher` whose first argument `buf` is an object type `bytes`, and the second argument `shift` is an integer. This function returns an object of class `bytes`, each element of which has a value that is the result of dividing by 256 sum of the corresponding element from `buf` and the number `shift`. In the function `main` test correct operation of the `caesar_cipher` function. For example, the result of calling `caesar_cipher(b'\x00\01\x02',3)` should be the object `b'\x03\x04\x05'`, while the result of calling `caesar_cipher(b'Python',-6)` should be the object `b'Jsnbih'`.
14. Write a program `cypher_cezara.py` similar to the program `cypher_xor.py`, but using function `caesar_cipher`.
15. Write a program that asks for a natural number in a loop and writes each given number to the appropriate file: even numbers to `even.txt`, and odd numbers to `odd.txt`. The program ends when 0 is entered. The files should be saved in a new folder that will be created when the program is started (see how much does not exist yet). To do this, you need to use the `os` module. Useful functions:
    - `os.path.exists(mypath)` - checks if a folder exists;
    - `os.mkdir(mypath)` - creates a folder in the given path `mypath`;
    - `os.getcwd()` - current directory.

    Running the program again should cause the new numbers to be added to the existing ones. files or new files will be created.
16. Write a program that reads the files created in the previous task and calculates the sum and arithmetic mean of the numbers contained in each of them.
17. There are three directories on the disk, and each of them contains three text files. Write a program that selects a directory at random, and then selects one of the files inside and prints its contents to the screen. We assume that the names of the directories and files are known to the user.
18. Write a program that, for a text file supplied by the user, displays statistics consisting of information about the number of lines it contains and all the words appearing in the file.
19. Write a program that creates an encrypted version of the file `data.txt` – any text file and saves it under the name `data.cez` (you can use the Caesar cipher).
20. Write a program that opens the file `in.txt`, then checks whether it contains the character strings `" ,"` (space, comma), converts them to `", "` (comma, space), and saves the result in the file `out.txt`.

Chapter 7

Object-oriented programming

From the very beginning of working with Python, we say that we work with objects, which are used to store and process data in computer programs. Each object has its `identity`, `type`, and `state`. Up until now, we have been using built-in types. Now, thanks to object-oriented programming, we will be able to reflect elements of the real world (and not only) by defining our own object types.

In this chapter, the reader will be introduced to the basic concepts of object-oriented programming, which will not only allow them to define their own data types, but also, by using closely related to this paradigm modularity, ensure better organization of the program.

7.1. Basic concepts

The most basic concept in object-oriented programming is `class`, which is a template describing how a given object will be constructed and how it will behave. An object created from a given class is called its `instance`, and the process of its creation `instantiation`. In a class, you can define `attributes` of the object, which are variables (class variables) and methods (static, class and instance). A `class variable` is a variable that is common to all instances of a class and is defined within the class but outside its methods. On the other hand, an `instance variable` (also called `field`) is a variable that belongs only to the current instance of the class and is defined inside methods.

The basic properties of object-oriented programming include:

– `Inheritance -` allows you to define new classes based on existing ones, extending or changing their functionality. Inheritance in Python is based on searching for attributes (in `X.name` expressions).

– `Polymorphism` - in the expression `X.method` the meaning of the attribute `method` depends on the type (class) of the object `X`.

– `Encapsulation` - methods and operators implement behavior and data hiding is the default convention.

7.1.1. Defining Classes

The `class` statement creates a class object and assigns it a name. It is an *executable* statement in Python (similar to `def`), meaning that when the Python interpreter encounters a `class` statement in your code, it executes it, generates a new class object, and assigns it the name given in the statement header. The `class` statement is normally executed the first time a file containing it is imported. Assignments inside a `class` statement create attributes of the class. In turn, top-level assignments inside a `class` statement (not embedded inside a `def` statement) generate attributes on the class object. Thus, the scope of the `class` statement becomes the namespace of the object's attributes, and they are accessed using the qualified syntax `obj.attribute`. Class attributes provide access to an object's state and behavior. They also record state information and behavior shared by all instances created from this class. `def` statements nested within `class` statements generate methods that process instances.

As an example, we will define a class representing a point in a coordinate system, which we present in listing 7.1.

Listing 7.1: *Class definition* `Point`

```python
# point.py
class Point:
  def __init__(self, x, y):
    self._x, self._y = x, y
  def move(self, deltaX, deltaY):
    self._x += deltaX; self._y += deltaY
  def __str__(self):
    return f"({self._x}, {self._y})"
```

The first argument of each method defined inside the `Point` class, named `self`, is a reference to the object for which this method will be called. Methods whose names start and end with two characters underscores `__` are `special` methods. For example, the `__init__` method is automatically called when creating an object of the class. This method is not, however, constructor of the class, but its `initializer`. The arguments of this method, except the first one, are used to initializing instance variables: `self._x` and `self._y`. Another special method in the body of the `Point` class is the `__str__` method, which creates a representation of the object in the form of the character string. As a result, displaying the object shows whatever it returns from its method `__str__`. To convert the `ob` object to a string, you can call the class `a` method `__str__`: `a.__str__()`. However, a more natural way is to use the built-in function `str` with an object as its argument: `str(ob)`. The `str` function will work as expected because it will call the method `__str__`. The `__str__` method is executed automatically every time the instance is converted to its chain display.

LISTING 7.2: *Program using an object of class* `Point`

```
1  # point_main_1.py
2  from point import Point
3  def main():
4    a = Point(5, 8)
5    print("Created point:", a)
6    a.move(-2, 3)
7    print("Point after displacement: ", a)
8    print("Type of created object:", type(a))
9    print("Dictionary __dict__ for the created object:")
10   print(a.__dict__)
11   b = Point(3, 2)
12   print("Point b:", b)
13 if __name__ == "__main__":
14   main()
```

How is `__repr__` different from `__str__`? Both return a string representation of a class object, and `__repr__` also tells you the name of the class the object is an instance of. We call it on an object, like this: `print(repr(a))`.

```
def __repr__(self):
return (f'{self.__module__}.'
        f'{self.__class__.__name__}'
        f'({self._x}, {self._y})')
```

Each time a class object is invoked, a new instance object is created and returned. Each instance object inherits the attributes of the class and gets its own namespace. Assignments to the `self` attributes in methods create attributes for individual instances. Assignments to the `self` attributes create or modify data in the instance, not in the class.

Instances of a class are created by calling a function whose name is the class name: `a = Punkt(5, 8)` (listing 7.2: line 5). After creating a new instance of the class, its attributes and methods are available using the member selection operator, i.e. the dot (`.`), e.g. `a.move(-2, 3)` (listing 7.2: line 7). Internally, each instance is implemented using `__dict__` dictionary containing unique information about this instances that can be printed to the screen as follows: `print(a.__dict__)` (listing 7.2: line 11).

In a class, we can also define a variable referring to the class, not the object, which will be shared by all objects of this class. To create a class variable, simply define it in the class body. To refer to it, we first use the class name, then a dot, and finally, the name of the variable. So there is no need to create any objects, although once they are created the variable can be called on their behalf. However, it is important to remember that when an instance variable of the same name is created when referring to the class variable through an object, Python will always identify the name of the instance variable, not the class variable. This can lead to errors. A class variable is often used, for example, to keep track of the number of objects of a given class that have been created, so when it is defined, it is assigned the value 0, and the special methods of the class initializer and destructor increment and decrement them by 1 accordingly. In the next part of the chapter, we will present the practical use of class variables - subsection 7.4.

7.1.2. Encapsulating names in a class

One of the basic properties of object-oriented programming is the encapsulation of data in objects of a given class, although Python itself does not provide mechanisms for this. There are mandatory naming conventions related to the purpose of data and methods. This convention states that any name starting with a single underscore (_) indicates an internal unit. Python does not block access to internal units. However, using them is considered inelegant and may lead to error-prone code, for example, by inadvertently removing attributes outside the class, as illustrated in listing 7.3 on line 7.

Listing 7.3: *Removing class attribute* `Point`

```
1  # point_main_2.py
2  from point import Point
3  def main():
4    a = Point(5, 8)
5    print("Created point:", a)
6    a._y
7    try:
8      print("After removing the _y attribute:", a)
9    except AttributeError as ex:
10     print("\n", ex.args)
11 if __name__ == "__main__":
12   main()
```

The fact that the `Point` class attribute has been removed, which we then try to refer to in the defined special method `__str__`, will be reported in a message displayed on the screen, and thanks to the use of exception handling, the program will end without being interrupted by its occurrence:

```
Point created: (5, 8)
After removing the _y attribute:
("'Punkt' object has no attribute '_y'",)
```

Most languages, to protect access to class members, define those members as private using the keyword `private` or similar. Private variables and methods are useful for two reasons:

1. increase code security and stability by selectively denying access to important or sensitive parts of an object's implementation; clearly define what is used internally by the class;
2. avoid naming conflicts resulting from inheritance; because they are private, each class has its own copies of them.

The names of instance variables and private methods in Python, by convention, start with double underscore, but they do not end with them. Neither a private variable nor a private method is visible outside the methods of the class in which they are defined. The privacy mechanism used distorts names of private variables and methods when the code is compiled to intermediate code by adding the class name with an underscore at the beginning of these names, e.g. using the `dir` function, which returns a list of attributes and methods of a given object, we can see the private attributes of the `Punkt` class instance (we deliberately do not show its other attributes and methods, focusing on those important to us):

```
>>> dir(Point(3, 5))
['_Point__x', '_Point__y', ...]
```

This operation is called "name mangling" and its purpose is to prevent any accidental sharing of the variable. You can of course deliberately "*simulate*" decoration that will take place and thus gain access to a variable, e.g. `a._Point__x = 111`. Decorating in the form indicated above makes it easier when debugging the program.

In listing 7.4, we present a version of the `Punkt` class definition in which instance variables have been marked as private. This makes it no longer possible to modify fields outside the class.

LISTING 7.4: *Class* `Point` *with private fields*

```
# point_priv.py
class Point:
  def __init__(self, x, y):
    self.__x, self.__y = x, y
  def move(self, deltaX, deltaY):
    self.__x += deltaX
    self.__y += deltaY
```

```
8   def __str__(self):
9     return f"({self.__x}, {self.__y})"
10
11 # main_priv.py
12 from pkt_priv import Punkt
13
14 def main():
15   a = Point(5, 8)
16   print("Created point:", a)
17   print(a.__dict__)
18   a.__x = 3
19   print("After statement a.__x = 3:", a)
20   print(a.__dict__)
21 if __name__ == "__main__":
22   main()
```

```
Point created: (5, 8)
{'_Point__x': 5, '_Point__y': 8}
After the statement a.__x = 3: (5, 8)
{'_Point__x': 5, '_Point__y': 8, '__x': 3}
```

In the case when we refer to the private instance variable (line 19), but without using the name decorator, it will only increase the dictionary `__dict__`, which is the namespace of the object containing its attributes, by the new attribute `'__x'`.

As mentioned above, each instance of a class has a dictionary `__dict__` associated with it that stores its attributes. This causes wasteful memory usage, especially for objects that have a small number of instance-level variables but a very large number of which are created in the program.

In classes that primarily function as simple data structures, you can often significantly reduce the amount of memory occupied by objects by adding an attribute to the class definition `__slots__`. It is a class-level variable that can be assigned the names of variables used by instances, which can be a string, an iterable, or a sequence of strings. In the following definition of class `Date`, the attribute `__slots__` is a tuple of the names of its private attributes, storing information about the year, month, and day.

```
# date.py
class Date:
  __slots__ = ('__year', '__month', '__day')
  def __init__(self, year, month, day):
    self.__year = year
    self.__month = month
    self.__day = day
```

When we define the `__slots__` attribute, Python will use a much more concise representation of objects. Instead of adding a dictionary to each object, Python then creates objects based on a small, fixed-size array, like a tuple or list. Attribute names listed in the `__slots__` specifier are internally mapped to specific array indices. A side effect of using this technique is that objects you cannot add new attributes - only those listed in the `__slots__` specifier are allowed. Although it may seem that the presented solution is useful in many situations, it should not be overused. In many places in Python, standard code is based on dictionaries. In addition, classes created using the described techniques do not support some mechanisms, e.g. `multiple inheritance`. Therefore, this technique should only be used in those classes that are often used in a program, e.g. when a program creates millions of objects of a given class. The slot technique is often treated as a tool providing airtightness, which prevents users from adding new attributes to objects.

For instances that do not have the `__dict__` attribute, it is not possible to assign variables that are not listed in the `__slots__` definition. Attempting to do so will result in an exception of type `AttributeError`.

```
d._a = 1
Traceback (most recent call last):
 File "mainDate.py", line 22, in <module> main()
 File "mainDate.py", line 9, in main d._a = 1
AttributeError: 'Date' object has no attribute '_a'
```

If the program requires the ability to dynamically add new variables, the name `'__dict__'` should be added to the string sequence assigned to the `__slots__` attribute. However, in this dictionary we will not see the attributes listed in `'__slots__'`.

```
#__slots__ = ('__year', '__month', '__day', '__dict__')
d._a = 1
print(d.__dict__)
"""output: {'_a': 1} """
```

Remember that the `__slots__` attribute is restricted to the class in which it is defined. Therefore, derived classes will include the `__dict__` attribute as usual, unless they also define `__slots__`.

7.2. Inheritance

Inheritance is a mechanism for creating new classes that extend already existing classes. The class from which we inherit is called the `base` or `superclass` (sometimes "parent"), and the inheriting class is called the `derived` or `subclass` (sometimes "child"). A subclass inherits all attributes and methods of the superclass, and a special case of an inherited method is the `__init__` initializer. The derived class initializer takes as arguments: first comes `self`, then arguments necessary to initialize attributes of the base class, and finally, arguments initializing attributes in the derived class. This order is conventional, but it allows for correctness checking. In the body of the initializer, we execute the original `__init__` method of the `Punkt` class, calling it via the class name and passing it `self` and the remaining arguments explicitly. Python uses inheritance to find and call only one `__init__` method at a time when creating an instance, the one lowest in the class tree. To execute the `__init__` method higher in the inheritance tree, we must call it manually via the name of the parent class. The advantage of this solution is that it leaves the programmer to decide whether and how to call the parent class initializer.

Inheritance is written in the `class` statement after the derived class name using a list enclosed in round brackets, in which the names of the base classes are listed, separated by commas. As an example, we define an extension of the class `Point` from listing 7.1 by defining a derived class `NamedPoint` (listing 7.5). This extension introduces an additional attribute storing the name of the point (line 7). In the derived class, we can override selected methods of the base class. As an example, we define in the derived class its initializer `__init__` (lines 5-7) and the method `__str__` (lines 8-9). On line 6 of the initializer definition, the base class initializer is explicitly called.

Listing 7.5: *Definition of derived class* `NamedPoint`

```
1 # namepoint.py
2 from pkt_priv import Punkt
3
4 class NamedPoint(Point):
5 def __init__(self, x, y, name):
6   Point.__init__(self, x, y)
7   self.__name = name
8 def __str__(self):
9   return f"{self.__name}" + Point.__str__(self)
```

In listing 7.6, we create the object `a` (line 6), as a point with coordinates `(5,8)`, which was given the name `"A"`. We print data about the created point to the screen and here, Python will select the `__str__` method redefined in the descendant class (line 7). This object inherited all of the attributes and methods of the `Point` class, so we move the point `a` in the coordinate system using the `move` method of the `Point` class (line 8). Again we print the values of the point `a` after moving it to the screen. On line 11 we create an object of class `Point` and print information about it to the screen (line 12).

Listing 7.6: *Defining base and derived class objects*

```
# Namepoint_main.py
from pkt_priv import Punkt
from namedpoint import NamedPoint

def main():
  a = NamedPoint(5, 8, "A")
  print('NamedPoint created:', a)
  a.move(-2, 3)
  print('NamedPoint after moving:', a)
  print('NamedPoint.__dict__:\n', a.__dict__)
  b = Point(3, 4)
  print('Point created:', b)

if __name__ == "__main__":
  main()
```

As a result of calling the program from listing 7.6, we will see the following information on the screen:

```
NamedPoint Created: A(5, 8)
NamedPoint after displacement: A(3, 11)
NamedPoint.__dict__:
{'_Point__x': 3, '_Point__y': 11, '_NamedPoint__name': 'A'}
Point Created: (3, 4)
```

The class `Point` is a subclass of itself:

```
>>> from point import Point
>>> from namedpoint import NamedPoint
>>> issubclass(Point, Point)
True
```

But it is not a subclass of `NamedPoint`:

```
>>> issubclass(Point, NamedPoint)
False
```

Whereas the class `NamedPoint` is a subclass of the class `Point`:

```
>>> issubclass(NamedPoint, Point)
True
```

Now, we create two objects: one of class `Point` and the other of class `NamedPoint` , for which we check the mutual dependencies.

```
>>> a = NamedPoint(5, 8, "A")
>>> b = Point(3, 5)
```

Is the object `a` an instance of the class `Point`?

```
>>> isinstance(a, Point)
True
```

Is the object `a` an instance of the class `NamedPoint`?

```
>>> isinstance(a, NamedPoint)
True
```

Is the object `b` an instance of the class `NamedPoint`?

```
>>> isinstance(b, NamedPoint)
False
```

Every Python class inherits directly or indirectly by the `object` class, and thus inherits its methods. Hence the header `classClassName` of the class definition is equivalent to the header `class ClassName(object)`. To check what the class `object` provides us, we can run the `dir` function in the interpreter for an object of this class, returning a list of its attributes.

```
>>> ob = object()
>>> dir(ob)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

The possibility of inheriting from subclasses gives us the ability to create rich hierarchies of arbitrarily specialized classes with a tree structure in which the edges are set "in reverse", i.e. the arrows lead from subclasses to their immediate base classes. In the case of the inheritance discussed in this chapter, this will be a tree of the form:

$$\texttt{Point} \leftarrow \texttt{NamedPoint}$$

When we call a method on an object of a given type, the method is searched for first in the class of that object, then in its base class (if any), then in its base class, and so on, visiting classes higher and higher in the inheritance hierarchy until the method is found or the classes are exhausted. In the example from listing 7.6 for object `a`, the method `move` will be searched for in the classes `NamedPoint, Point`, and will be found in the class `Point`. In the case of the method `__str__`, it will already be found in the class `NamedPoint`, hence the need to define in its body a call to the appropriate version of it from the base class. This mechanism allows for the creation of arbitrarily specialized classes of objects, performing individual operations in their own way, which can also affect the implementation of algorithms (operations) specified in their superclasses. All this while maintaining a strict type hierarchy, allowing for easy verification of what operations are available for a specific object.

We can achieve the same effect using the built-in `super` object:

```
class NamedPoint(Point):
def __init__(self, x, y, name):
  super().__init__(x, y)
  self.__name = name
def __str__(self):
  return f"{self.__name}"+ super().__str__()
```

A `super` object created in the implementation of a class method will behave just like `self`, but for such an object the order of searching for methods will be different: they will be searched for starting from the base class of the given class. In this situation, `super().__str__()` tries to search for `__str__` methods starting from the class `Point`, not `NamedPoint`, and call it for the `self` parameter. A class can inherit from more than one class, which is noted in the header of the `class` statement, listing each superclass in parentheses and separating them with commas. In this type of inheritance, the order in which the superclasses are listed in the header of the `class` statement can be crucial. Multiple inheritance is a very advanced topic in Python programming and is beyond the scope of this manual. The interested reader is referred to the Python documentation or the books listed in the reference list of this manual.

The use of the `super` function is more relevant to single inheritance trees and starts to cause problems as soon as traditionally coded classes start to use multiple inheritance. In single inheritance mode, the function can mask later problems and as the tree grows, result in unexpected behavior. The `super` function will not throw an exception in a multiple inheritance tree, but will naturally choose the leftmost parent class that has a method to call, which may or may not be the desired method. In such a case, it is recommended to explicitly choose the parent class.The `super` function should be used after deep analysis and understanding of advanced Python programming concepts.

7.3. Static and class methods

A `class` method is a function defined in the scope of a class, but preceded by the decorator `@classmethod`, whose first argument is conventionally called `cls` and is a reference to the class. An `instance` method (short: `method`) is a function defined in the scope of a class, and its first argument is customarily called `self` and is a reference to the object on which the method is called. Additionally, there are methods in a class that do not operate on a specific instance of the class, which are called `static`. In Python they do not have a `self` parameter, but they are provided with `@staticmethod` decorator. A static method can be called either by the class name or by its object, and in both cases, the result will be the same itself. Technically, it is just a regular function placed after just in class scope instead of global scope.

In listing 7.7, we present an example of a `Date` class representing a date with the following fields: year (`y`), month (`m`), and day (`d`). Its body also defines two static methods. The `today` method converts the current local time stamp to the format of the `Date` class. In turn, the second method `tomorrow` converts the current local time stamp after increasing it by one day to the format of the `Date` class.

LISTING 7.7: *Example of class* `Date` *with static methods*

```
# date.py
import time

class Date(object):
  def __init__(self, y, m, d):
    self._y, self._m, self._d = y, m, d

  @staticmethod
  def today():
    t = time.localtime()
    return Date(t.tm_year, t.tm_mon, t.tm_mday)

  @staticmethod
  def tomorrow():
    t = time.localtime(time.time() + 24 * 60 * 60)
    return Date(t.tm_year, t.tm_mon, t.tm_mday)
```

Next, we define a derived class `PolishDate` (listing 7.8) inheriting from the `Date` class, which has a method that formats the date to the form `'yyyy-mm-dd'`.

LISTING 7.8: *Example of a derived class* `PolishDate` *from a class* `Date`

```
#polishdate.py
from date import Date

class PolishDate(Date):
  def __str__(self):
    s = "{:04}-{:02}-{:02}"
    return s.format(self.y, self.m, self.d)
```

Listing 7.9: *Example of using objects of class* `PolishDate`

```
#mainDate.py
from polishdate import PolishDate

def main():
  d = PolishDate.today()
  print(type(d))
  print("Today is", d)

if __name__ == '__main__':
  main()
```

```
<class 'date.Date'>
Today is <date.Date object at 0x7f0ebebab6a0>
```

The results of running the program in listing 7.9 are not the same as what we would like and should expect. This is because the methods placed in the `Date` class `today` and `tomorrow` are static methods. To output a properly formatted date to the screen, the above two methods should be class methods. However, a useful static method in the `Date` class would be `seconds_per_day` method, which returns the number of seconds in a day and can be used in class methods.

```
class Date:
  ...
  @staticmethod
  def seconds_per_day():
    return 24 * 60 * 60
```

Class methods are called for the entire class, not just for a specific class. its instance and take that class as their first parameter. This argument is often called `cls`, but it is much a weaker convention than `self`. In order to distinguish them from other types, class methods are marked with the `@classmethod` decorator. Like static methods, they can be called in two ways ways – using a class or an object – but in both cases, only a class will be passed to `cls`. In general, it can also be a derived class.

Listing 7.10 presents an improved version of the `Date` class using class methods, thanks to which, when the program from listing 7.9 is called, the date will appear on the screen in the correct and expected format.

```
<class '__main__.PolishDate'>
Today is 2024-06-25
```

LISTING 7.10: *Class* `Date` *with class methods*

```
# date.py
from time import localtime
class Date(object):
  def __init__(self, y, m, d):
    self._y, self._m, self._d = y, m, d

  @classmethod
  def today(cls):
    t = localtime()
    return cls(t.tm_year, t.tm_mon, t.tm_mday)

  @classmethod
  def tomorrow(cls):
    t = localtime(time.time() + seconds_per_day())
    return cls(t.tm_year, t.tm_mon, t.tm_mday)

  @staticmethod
  def seconds_per_day():
    return 24*60*60;
```

7.4. Operator Overloading

Classes allow for `operator overloading`, which involves capturing and implementing the behavior of operations on built-in types by defining methods with special names in the class body. Each of these methods starts and ends with a double underscore. These names are not reserved and can be inherited from classes superordinates in the usual way. Python finds and calls at least one such method in every operation, and does this automatically when instances are found in expressions and other contexts. In classes, it is allowed to mix methods processing numbers and collections and mutable operations and immutable. Most operator overloading names have no value defaults, and their corresponding actions throw an exception if the specified method is not defined. Operator overloading should be done in a class only when it is absolutely necessary to maintain the consistency of operations as for built-in types. Otherwise, we recommend using regular methods. Operator overloading is most often used when defining mathematical structures.

In this section we will show how to overload selected built-in operators for a user-defined class. As an example, we will define a class `Vector`, representing `n`-dimensional vector which is a list of `n` integers representing its components (i.e. `[x_1,...,x_n]`, where the numbers `x_i` are the vector's components) and basic operations performed on it.

We will start by defining a class initializer `Vector`, which initially creates an empty list of its components (`_vect`) and copies to it the elements of the list `list` passed as the third argument. The field `_size` is assigned the value passed as the second argument if it is equal to the length of the passed list. Furthermore, within the definition of the class `Vector`, a class variable `_ile` was defined, which will continuously monitor the number of created class instances.

Listing 7.11: *Class definition* `Vector`

```python
class Vector:
  _how much = 0
  def __init__(self, size = 0, list = []):
    if (n:=len(list)) != size:
      self._size = n
    else:
      self._size = size
    self._vect = []
    for and in list: self._vect.append(i)
    Vector._ile += 1
```

7.4.1. Methods for comparing objects

`Rich comparison methods` are invoked on all expressions that use comparisons. These include:

```
__lt__(self, other) # self < other
__le__(self, other) # self <= other
__eq__(self, other) # self == other
__ne__(self, other) # self != other
__gt__(self, other) # self > other
__ge__(self, other) # self >= other
```

The above methods can return any value, but if the comparison operator is used in the context of the operation logical, the returned value will be interpreted as the logical result (of type `bool`) of the operator's action. These methods can also return (though they do not throw an exception) a special object `NotImplemented` in case the operands do not support them. The effect is as if the method had not been defined at all. There are no implicit relationships between comparison operators; for example, the fact that `x == y` evaluates to `True` does not mean that `x != y` automatically evaluates to `False`. To make the operators work symmetrically, a method must be defined `__ne__` together with the `__eq__` method. There are also no right-handed ones (with swapped arguments) versions of these methods to be used in situations where the left the argument does not support the specified action, and the right one does support. The methods `__lt__` and `__gt__`, `__le__` and `__ge__`, and `__eq__` and `__ne__` are reflections of each other. In Python 3.x, for sorting operations, you should use `__lt__` methods.

Two vectors are different if their dimensions are different otherwise if they have different components.

```
def __ne__(self, other):
  if self._size != other._size:
    return True
  for i in range(self._size):
    if self._vect[i] != other._vect[i]:
      return True
  return False
```

7.4.2. Basic methods of binary operations

If any of the following binary operation methods do not support the operation of the arguments passed, then they should return (not report) built-in object called `NotImplemented`, which acts as if the method was not defined at all.

The basic methods of two-argument operations include:

`__add__(self, other) # self + other`

Performs addition of numbers or concatenation of sequences.

```
def __add__(self, other):
  if !isinstace(other,Vector):
    return NotImplemented
  if self._size != other._size:
    raise ValueError('Different vector sizes!')
  V = Vector(self._size, self._vect)
  for i in range(self._size):
    V._vect[i] += other._vect[i]
  return V
```

`__sub__(self, other) # self - other`
`__mul__(self, other) # self * other`

Performs multiplication of numbers or repetition (duplication) of sequences.

```
def __mul__(self, alpha):
  V = Vector(self._size)
  for i in range(self._size):
    V._vect.append(self._vect[i])
```

```
    for i in range(self._size):
      V._vect[i] *= alpha
    return V
```

`__truediv__(self, other) # self / other`

To perform division (taking into account the remainder).

`__floordiv__(self, other) # self // other`

In order to perform division with truncation (integer part of the division).

```
__mod__(self, other)  # self % other
__divmod__(self, other) # divmod(self, other)
__pow__(self, other])  # pow(self, other) | self **other
__lshift__(self, other) # self << other
__rshift__(self, other) # self >> other
__and__(self, other)  # self & other
__xor__(self, other)  # self ^ other
__or__(self, other)  # self | other
```

7.4.3. Right-side binary operations methods

The names of right-hand equivalents of double-argument operators, described in subsection 7.4.2, start with the prefix `r`, e.g. the right-hand equivalent of the `__add__` operator is `__radd__`. The right-handed varieties have the same lists arguments, but the argument `other` is on the left operator. For example, the operation `self + other` calls the method `self.__add__(other)`, while `other + self` calls `self.__radd__(other)` method.

The right-hand side methods `(r)` are only called when the class instance is on the right side and the left operand is not an instance of the class that implements the operation:

`item + otherobject` runs the `__add__` method
`item + item` runs the `__add__` method
`otherobject + instance` runs the `__radd__` method.

If there are objects of two overloading classes in the activity action, then the class of the argument that comes after is preferred left side.

The `__radd__` method is often implemented like this: swaps the order of operands and calls the `__add__` method. The right-hand side methods of binary operations include:

`__radd__(self, other)  # other + self`

```
def __radd__(self, other):
  if isinstance(other,int):
    V = Vector(self._size)
    for i in range( self._size):
      V._vect.append(self._vect[i])
    for i in range(self._size):
      V._vect[i] += other
    return V
  elif isinstance(other, list):
    V = Vector(len(other), other)
    return V + self
  else:
    return NotImplemented
```

```
__rsub__(self, other)  # other - self
__rmul__(self, other)  # other * self
__rtruediv__(self, other) # other / self
__rfloordiv__(self, other) # other // self
__rmod__(self, other)  # other % self
__rdivmod__(self, other) # divmod(self, other)
__rpow__(self, other)  # pow(other, self) | other**self
__rlshift__(self, other) # other << self
__rrshift__(self, other) # other >> self
__rand__(self, other)  # other & self
__rxor__(self, other)  # other ^ self
__ror__(self, other)  # other | self
```

7.4.4. Dual-argument methods with in place update

The methods of binary operations with in-place update are called for the following assignment statement formats: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=` and `|=`. These methods should attempt to perform the action in place (with modification of the `self` instance) and return the result, which could be an instance of `self`. If the method is undefined, the update operation falls back to the regular methods. To evaluate the expression `X += Y`, where `X` is an instance of the class with the defined method `__iadd__`, the `x.__iadd__(y)` method is called. Otherwise, the `__add__` and `__radd__` methods are used.

The update-in-place assignment methods are:

```
__iadd__(self, other)  # self += other
```

```
def __iadd__(self, other):
  if self._size == other._size:
    for i in range(self._size):
      self._vect[i] += other._vect[i]
    return self
  else:
    raise ValueError('Different vector sizes!')
```

```
__isub__(self, other)  # self -= other
__imul__(self, other)  # self *= other
__itruediv__(self, other) # self /= other
__ifloordiv__(self, other) # self //= other
__imod__(self, other)  # self %= other
__ipow__(self, other)  # self **= other
__ilshift__(self, other)  # self <<= other
__irshift__(self, other)  # self >>= other
__iand__(self, other)  # self &= other
__ixor__(self, other)  # self ^= other
__ior__(self, other)  # self |= other
```

7.4.5. Other selected methods of action

The method `__del__` of class `object` is a `destructor`. It is used to destroy objects and for an object of class `Vector` it is called as follows: `del V`. The definition of the destructor for class `Vector`, whose task is to decrease by 1 the value of the class variable `_ile`, has the form:

```
def __del__(self):
  Vector._how many -= 1
```

In the case of a class representing a vector, two more frequently used operations will be needed: `__getitem__` which retrieves the value of the field that is a list located at the index indicated in the argument, and `__contains__`, whose task is to check whether the value given as an argument is an element of the field that is a list.

```
def __getitem__(self, i):          # self[i]
  return self._vect[i]

def __contains__(self, element):  # element in self
  return element in self._vect
```

7.4.6. Example of class `Vector`

We present the definition of the `Vector` class with selected methods of this class described in the previous subsections defined.

```
# vector.py

class Vector:
  _how_many = 0

  def __init__(self, size = 0, a_list = []):
    if (n:=len(a_list)) != size:
      self._size = n
    else:
      self._size = size
    self._vect = []
    for i in a_list:
      self._vect.append(i)
    Vector._how_many += 1

  def __del__(self):
    Vector._how_many -= 1

  def get_how_many(cls):
    return cls._how_many

  def length(self):
    return self._size

  def __add__(self, other):
    if self._size != other._size:
      raise ValueError('Different vector sizes!')
    V = Vector(self._size, self._vect)
    for i in range(self._size):
      V._vect[i] += other._vect[i]
    return V

  def __radd__(self, other):
    if isinstance(other,int):
      V = Vector(self._size)
      for i in range( self._size):
        V._vect.append(self._vect[i])
```

```
      for i in range(self._size):
        V._vect[i] += other
      return V
    elif isinstance(other, list):
      V = Vector(len(other), other)
      return self + V
    else:
      return NotImplemented

  def __iadd__(self, other):
    if self._size == other._size:
      for i in range(self._size):
        self._vect[i] += other._vect[i]
      return self
    else:
      raise ValueError('Different vector sizes!')

  def __mul__(self, alpha):
    V = Vector(self._size)
    print('mul=',self._size)
    for i in range( self._size):
      print(self._vect[i])
      V._vect.append(self._vect[i])
    for i in range(self._size):
      V._vect[i] *= alpha
    return V

  def sum(self):
    return sum(self._vect)

  def __getitem__(self, i):
    return self._vect[i]

  def __ne__(self, other):
    if self._size != other._size:
      return True
    for i in range(self._size):
      if self._vect[i] != other._vect[i]:
        return True
    return False


```

```
  def __contains__(self, element):
    return element in self._vect

  def __str__(self):
    s = '['
    for i in self._vect[:-1]:
      s += str(i) + ', '
    s += str(self._vect[-1]) + ']'
    return s
```

```
# main_Vector.py
from vector import Vector

def main():
  print('Number of Vector objects:', Vector._how_many)
  V = Vector(5, [1,2,3,4,5])
  print('V =', V)
  V1 = Vector(4, [1,2,3,4])
  print('V1 =', V1)
  try:
    In = V + V1
  except ValueError as in:
    print(in)
  print('5 + V:', 5 + V)
  print('[1,1,1,1,1] + V:', [1,1,1,1,1] + V)
  try:
    print('2.2 + V:', 2.2 + V)
  except:
    print('OOPS! Something went wrong!')
  V+=V
  print('V+=V:', V)
  V1 = Vector(5, [1,2,3,4,5])
  print('V1 =', V1)
  try:
    In = V + V1
  except ValueError as in:
    print(in)
  print('After +:', W._size)
  print('After +:', W._vect)
  W = W * 3
  print('W = W * 3:', W)
  print('sum of W:', W.sum())
```

```
  print('W[2] =', W[2])
  #print('W[12] =', W[12])
  print('18 in W?', 18 in W)
  print('1 in W?', 1 in W)
  print('W != W?', W != W)
  print('V =', V)
  print('W =', W)
  print('V != W?', V != W)
  print('Number of Vector objects:', V.get_how_many())
  del V
  print('Number of Vector objects:', Vector._how_many)
if __name__ == "__main__":
  main()
```

7.5. Properties

Python allows programmers to directly access instance variables, without the need to create instrumentation in the form of `getters` and `setters`, common in other languages object-oriented. The lack of methods for setting and reading values makes the class code in Python cleaner and simpler, but in some situations using `getters` and `setters` can be convenient. Let's assume we would like to perform some operation on the value before we assign it to an instance variable. Or it would be useful to calculate the value of a variable on the fly. In both cases, methods like `get` and `set` would be the response to this demand, but their cost would be a loss Python's characteristic ease of access to variables instance.

The solution here is to use `property` which combines the ability to access an instance variable via `getters` or `setters` and the clarity of Python's instance variable notation. To create a property, we need the `@property` decorator and methods of type `get` with the same name as the property. Without a function that provides a way to set the value, such a property is read-only. In order to change it, you need a method of type `set`.

LISTING 7.12: *Class* `NemeVector` *using properties*

```
#namevector.py
class NameVector(Vector):
  def __init__(self, size, list, name):
    Vector.__init__(self,size,list)
    self.__name = name

  @property
  def name(self):
    return self.__name
```

```
  @name.setter
  def name(self, new_name):
    self.__name = new_name

  def __str__(self):
    s = self.__name + ' = ['
    for i in self._vect[:-1]:
      s += str(i) + ', '
    s += str(self._vect[-1]) + ']'
    return s
```

Listing 7.13: *Using the class property* `NemeVector`

```
# main_namevector.py
from vector import Vector
from namevector import NameVector

def main():
  nV = NameVector(5,[11,12,13,14,15],"vector")
  print(nV)
  print(nV.name)
  nV.name = "Vec"
  print(nV)
  print(type(nV))
  del nV

if __name__ == "__main__":
  main()
```

As a result of executing the program from listing 7.13, the following information will appear on the screen:

```
vector = [11, 12, 13, 14, 15]
vector
Vec = [11, 12, 13, 14, 15]
<class 'vector.NameVector'>
```

7.6. Serializing Python Objects

The `pickle` module allows you to serialize and deserialize Python objects. `serialization` (also known as *marshalling*, *pickling*, or *flattening*) is the process of converting an object into string bytes. `Deserialization` (also called *demarshalling* or *unpicking*) is the opposite process - replacing subsequent bytes per object. The sequence of bytes obtained in this way can be saved to a file or sent over the network. The data saved in the file can later be used to restore the state of the program the next time it is run.

The `dumps` function allows you to serialize an object to a sequence of bytes or otherwise allows you to save an object in a string. A sequence of bytes can be saved to a file or sent over the network.

```
import pickle

phone_book = {"Joan": "542124", "Mathew": "542323"}

bytes = pickle.dumps(phone_book)
print(bytes)

"""output:
b'\x80\x04\x95(\x00\x00\x00\x00\x00\x00\x00}
\x94(\x8c\x05Joan\x94\x8c\x06542124\x94\x8c
\x06Mathew\x94\x8c\x06542323\x94u.'
"""
```

We can save data to a file using the `dump` function. The file in which we want to save the data must be opened in binary mode. Please also remember to close it.

```
import pickle

phone_book = {"Joan": "542124", "Mathew": "542323"}
with open('app_data.pickle', 'wb') as file:
  pickle.dump(phone_book, file)
```

The `loads` function allows you to convert a sequence of bytes to an object (allows you to play from a string).

```
import pickle

bytes=(b'(dp0\nVJoan\np1\nV542124\np2\nsVMathew\np3\nV542323\np4\
    ns.')

phone_book = pickle.loads(bytes)
print(phone_book)

"""output: {'Joan': '542124', 'Mathew': '542323'} """
```

We can read data from a file using the `load` function. The file from which we want to read data must be opened in binary mode. We should also remember to close it.

```
import pickle

with open('app_data.pickle', 'rb') as file:
  phone_book = pickle.load(file)
  print(phone_book)

"""output: {'Jonna': '542124', 'Maciej': '542323'} """
```

In most programs, the `dump` and `load` functions are sufficiently using the `pickle` module effectively. This solution works for most Python data types and classes defined by users. If we use a library that allows saving and reproducing Python objects in databases or uploading objects over the network, it is very possible that it uses a module `pickle`. The `pickle` module is responsible for Python-specific self-descriptive data encoding. Because it is self-descriptive, serialized data contains information about the beginning and end of each object and its type. Therefore, you do not have to worry about defining records - code works without it.

```
>>> import pickle
>>> f = open("somedata", "wb")
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump("Hello", f)
>>> pickle.dump({"Apple", "Pear", "Banana"}, f)
>>> f.close()
>>> f = open("somedata", "rb")
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'Hello'
>>> pickle.load(f)In the data_stream there is
the function call is made
system(), which starts
given command in the console.
{'Apple', 'Pear', 'Banana'}
```

This way you can serialize functions, classes and objects, where only references are encoded in the generated data to related objects from code. Here's an example:

```
import math
import pickle
print(pickle.dumps(math.log))
print(pickle.dumps([math.sin, math.cos]))
```

At the time of deserialization, the program assumes that all the necessary source code is available. Modules, classes, and functions are automatically imported as needed. When Python data is shared across interpreters from different computers, this can make code maintenance difficult, because all computers need to have access to the source code itself. The `pickle.load` function should never be used for untrusted data. As part of the code loading, the `pickle` module automatically takes modules and creates objects based on them. An attacker who knows how the `pickle` module works can prepare specially crafted data that causes Python will execute certain system commands. Therefore, the `pickle` module should only be used internally in interpreters that can authenticate each other.

```
# There is a function call in the data_stream
# system(), which runs the given command in the console.
import pickle
bytes = b"cos\nsystem\n(S'ls -la /'\ntR."
pickle.loads(bytes)

"""output
total 2097236
drwxr-xr-x 19 root root 4096 Mar 17 2022 .
drwxr-xr-x 19 root root 4096 Mar 17 2022 ..
lrwxrwxrwx 1 root root  7 Mar 17 2022 bin -> usr/bin
drwxr-xr-x 4 root root 4096 Jun 18 2022 boot
drwxr-xr-x 2 root root 4096 Mar 17 2022 cdrom
drwxr-xr-x 20 root root 4560 Mar 25 18:01 dev
"""
```

Some objects cannot be serialized this way. These are usually objects that have an external state in the system, such as open files, open network connections, threads, processes, stack frames etc. In user-defined classes, you can sometimes workaround this limitation by providing `__getstate__` methods and `__setstate__`. Then, the `pickle.dump` function calls the `__getstate__` method to retrieve the serialized object, and when deserializing it is called, there is the `__setstate__` method. To illustrate the possibilities of this approach, the next slide shows a class with an internally defined thread that can be both serialized and deserialized (as the file `countdown.py`).

```
import time, threading
class Countdown:
  def __init__(self, n):
    self.n = n
    self.thr = threading.Thread(target=self.run)
    self.thr.daemon = True
    self.thr.start()
```

```
    def run(self):
        while self.n > 0:
          print('T-minus', self.n)
          self.n -= 1
          time.sleep(5)
    def __getstate__(self):
      return self.n
    def __setstate__(self, n):
      self.__init__(n)
```

```
>>> import pickle
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # After some time
>>> f = open("cstate.p", "wb")
>>> pickle.dump(c, f)
>>> f.close()
```

Now we can exit the Python interpreter and re-enter it run, call the following code:

```
>>> import pickle
>>> f = open("cstate.p", "rb")
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

We see the thread magically starts working again and resumes work from where it left off at the time of serialization.

The `pickle` module does not provide high coding efficiency large data structures, e.g. binary arrays created by libraries such as the `array` or `numpy` module. If we want to transfer large amounts of table data, it is better the solution may be to save them in a file or use standard encoding, e.g. `HDF5` (supported by custom libraries https://docs.h5py.org/en/stable/). Since the `pickle` module only works in Python and requires code source, it should not normally be used for long-term data storage. If the source code is modified, all stored data may become unreadable.

When storing data in databases or archives, it is usually better to use more standard encodings, e.g. XML, CSV, or JSON. They are more standardized and supported by many languages and are better adapted to changes in the source code. Also, it's worth remembering that the `pickle` module provides a lot of different options and has complicated edge cases. When performing typical tasks, you don't need to worry about them. However, if we are working on a complex application that serialization uses the `pickle` module, please read its official documentation: https://docs.python.org/3/library/pickle.html

We encourage the reader to familiarize themselves with other Python modules for working with data stored in binary files, such as the `shelve` module implementing the BSD database interface (*Berkeley Software Distribution Database Interface* (https://docs.python.org/3/library/shelve.html). This is an easy-to-use module for storing and reading data from a file on disk, ideal for less complex applications that need an easy way to permanently store Python data structures.

Due to the use of the `pickle` module, often automatically, in other modules for working with data stored on disk, which is the case, among others, in the previously mentioned `shleve` module, we have limited ourselves in this manual to describing only this module.

7.7. Practice exercises

1. Implement the `Address` class to store information about: house number, street, optionally apartment number, city, and zip code. Define an initializer so that the object can be created in one of two ways: with or without an apartment number. Provide a `show` method that prints an address with the street on one line and the zip code and city on the next line.
   Provide a `comesBefore(self, other)` method that checks if a given address comes before another when comparing by zip code.
2. Implement the `Car` class with the following properties. A car has a certain fuel efficiency (measured in kilometers/liter) and a certain maximum amount of fuel in the tank. The capacity is specified in the constructor and the initial fuel level is 0. Deliver `drive` method, which simulates driving a car for a certain distance, decreasing the fuel level in the tank, `getFuelLevel` method, which returns the current fuel level, and the `addFuel` method, which simulates refueling, but the maximum tank capacity cannot be exceeded. Example usage:

```
my_car = Car(20,40) # Efficiency 20 km/liter, tank capacity 40
my_car.addFuel(30)  # Fill up to 30 liters
my_car.drive(100)   # Drive 100 m
print(my_car.getFuelLevel()) # Print the amount of fuel left
```

3. Implement the `Student` class. For this exercise, a student has a first and last name and a total quiz score. Define an appropriate initializer and methods `getName()`, `addQuiz(score)`, `getTotalScore()`, and `getAverageScore()`. To calculate the average score across all quizzes using these methods, you must store the number of quizzes a student took in the appropriate field.
4. Create a file `dog.py` in which you define and test a class `Dog` representing a dog that has:
   - `name` - default `'Rex'`,
   - `toys_you_like` - your favorite toys - by default a list with only one element `bone`, the items on this list are not repeated,
   - `toys_disliked` - toys that the dog does not like - by default a list with only one item `dog`, items on this list do not repeat,

   in addition, the dog can:
   - greet and introduce yourself,
   - tell what his/her favorite/dislikeable toys are and how many of them there are,
   - like a toy, e.g. a new one or one you haven't liked before,
   - stop liking a toy.

   A dog may only like a new toy it encounters or pass it by indifferently. Think about how to remember a new toy that the dog will pass by indifferently. Important: the lists of liked and disliked toys have no common elements.
5. Using the `Dog` class, create a simulation of a dog walking in a circle, which should pass many boxes with random, but one kind of toys in one box. Additionally, we assume that the contents of the boxes may be repeated. When the dog encounters a box, it sees the toys kept there and can:
   - pass by the box indifferently,
   - randomly like one of the toys or choose the one he likes,
   - stop liking a selected toy if it was previously his favorite.

   The dog continues its journey until it reaches a predetermined number of repetitions (e.g. `100`) or at least one box is empty. After the journey is over, the dog shows the toys it likes and dislikes. For simplicity, we assume that the dog encounters random boxes in each iteration. The toy lists (boxes) are initially read from a file that you have to create yourself and save on separate lines, separated by a space, by the names of toys of the same type.
6. Extend the definition of the `Vector` class discussed in subsection 7.4.6 with definitions overloading those operators that are necessary for its proper functioning.
7. Write a class `Rational` representing the rational numbers `p/q`. The numbers `p` and `q` should be remembered as relatively prime with positive `q`. Implement:
   (a) An initializer with two integer arguments, numerator and denominator, where

the default value of the numerator should be zero and the denominator should be one. The initializer should work correctly even if the arguments given are not relatively prime or the denominator is negative.

(b) Member functions `getNumerator` and `getDenominator` that return the numerator and denominator of a number, respectively.

(c) The `__repr__` member function that returns a string representing a rational number.

(d) A member function `__float__` that returns a value of type `float` corresponding to a given rational number.

(e) Member functions `__add__` and `__sub__`.

(f) Member functions `__eq__, __ne__, __lt____le____gt____ge__`.

In the `main` function, read the numerator and denominator for two rational numbers, create from the read two rational numbers, and then print out the results obtained by applying the defined operators on the following lines.

8. Extend the class definition from the previous task by defining member functions `__mul__` and `__truediv__`.
9. Extend the class definition from the previous task by defining a function that implements unary minus.
10. Implement the `__eq__` function in a way that takes advantage of the fact that two numbers are equal if and only if neither is less than the other.
11. Using the `__slots__` attribute implement the `Punkt` class in the `punkt.py` file with the `__x` and `__y` properties. For both properties, define `setter` and `deleter`. Also define the special methods `__repr__` and `__str__` as shown presented in the chapter. Test the `Punkt` class in the `main` function defined in the file `test_punkt.py`.
12. Using the `__slots__` attribute, implement the `NamedPoint` class that inherits from the `Point` class. Test the `NamedPoint` class in the `main` function defined in the `test_namedpoint.py` file.
13. Write a program in which, using the `Point` and `NamedPoint` classes from previous tasks, create a list `points` with four objects of these classes (two objects from each class). Using the `pickle` module, save the `points` list in the file `points.pkl`.
14. Trees grow in a newly planted forest. Each tree is planted by a human and initially has a diameter of 1 meter. Suddenly, a beaver starts roaming the forest, which goes hunting every day and nibbles a random tree. All the other trees grow a little every day - their diameter increases by `1%`. The nibbled tree decreases its diameter by as much as `0.2m` and can only grow the next day. The beaver, which initially has strength equal to `2`, gets tired of nibbling trees, and its strength decreases by an amount equal to `7%` of the tree's diameter in meters. Check after how many days either the beaver exhausts its strength or the beaver knocks down the tree when

its diameter decreases to zero. Define the classes `Beaver`, `Tree` and `Forest` and the main function `main` simulating the actions described above.

Chapter 8

Advanced Python Elements

This chapter is about advanced Python data techniques that allow you to effectively manage and manipulate collections and sequences of data. You'll learn about list comprehensions, iterators, generators, enumerations and how to use them to improve the performance of your programs, as well as to make your code more readable and smaller.

8.1. List comprehension

List comprehension is a construction that allows you to create concise lists. A typical use is to create a list whose elements are the result of applying a certain operation to each element of another sequence or iterable object. Another typical use is to create a list of these sequence elements that meet certain conditions. A comprehensible list is created in square brackets, in which there is an expression followed by a `for` clause, followed by zero or more `for` or `if` clauses.

We will now present some examples of creating comprehensible lists:

1. list of squares of 10 initial natural numbers;

   ```
   squares = [x**2 for x in range(1,11)]
   ```

2. list of string characters, omitting spaces and digits;

   ```
   characters = [c for c in string if not (c.isdigit() or c.isspace())]
   ```

3. list of pairs with different numbers.

   ```
   pairs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
   ```

In listing 8.1, the following list comprehensions are created:

line 3: duplicate sequence values `vector`;

line 5: only positive values of the sequence `vector`;

line 7: absolute values from sequence values `vector`;

line 10: strings of the sequence `basket` with characters converted to uppercase letters;

line 12: pairs consisting of numbers from the range `range(1,6)` and their squares;
line 15: flattens the matrix `vector` into a one-dimensional sequence of its elements.

LISTING 8.1: *Examples of creating foldable lists*

```
1  def main():
2    vector = [-4, -2, 0, 2, 4]
3    doubled =[x * 2 for x in vector]
4    print(double)
5    only_positive = [x for x in vector if x > 0]
6    print(only_positive)
7    absolute_values = [abs(x) for x in vector]
8    print(absolute_values)
9    basket = ['apple','pear','plum','mango','banana']
10   basket = [fruit.upper() for fruit in basket]
11   print(basket)
12   pairs = [(x, x**2) for x in range(1,6)]
13   print(pairs)
14   vector = [[1,2,3], [4,5,6], [7,8,9]]
15   vector = [num for elem in vector for num in elem]
16   print(vector)
17
18 if __name__=='__main__':
19   main()
```

As a result of running the program from listing 8.1 we will see on the screen:

```
[-8, -4, 0, 4, 8]
[2, 4]
[4, 2, 0, 2, 4]
['APPLE', 'PEAR', 'PLUM', 'MANGO', 'BANANA']
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If we want to use the conditional statement `if` with the clause `else` in a list comprehension, we must place it before the `for` loop. Listing 8.2 shows an example using the above construction to create a list of triplets of positive values of the sequence `vector` and the squares of its elements for negative values or zero.

LISTING 8.2: *List of foldables with the construction* `if-else-for`

```
def main():
  vector = [-4, -2, 0, 2, 4]
  if_else_for = [x*3 if x > 0 else x**2 for x in vector]
```

```
  print(if_else_for)
if __name__=='__main__':
  main()
```

The starting expression in a list comprehension can be any expression, this one containing another list comprehension. Listing 8.3 shows a program that uses nested lists to construct the transposed matrix (line 7) of the matrix `matrix`.

Listing 8.3: *Creating a transposed matrix*

```
1 def main():
2   matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
3   for i in matrix:
4     for j in i:
5       print("{0:4}".format(j),end="")
6     print()
7   A = [[row[col] for row in matrix] for col in range(4)]
8   print()
9   for and in A:
10     for j in i:
11       print("{0:4}".format(j),end="")
12     print()
13 if __name__=='__main__':
14   main()
```

We advise you to use the appropriate built-in function whenever possible rather than creating complex nested list constructs.

One of such built-in functions is the `zip` function, which takes corresponding elements from one or more iterable objects and creates tuples from them until the shortest iterable object is exhausted. However, if we want the resulting structure of the type `zip object` to be a list, we must use the `list` function for the data structure created by the `zip` function.

For example, let's consider two lists: `leaders`, where the names and surnames of famous people, leaders in the IT industry, are stored, and `id`, which contains integer identifiers from 1 to 4. Applying the `zip` function to these sequences, we obtain a list of pairs of related information using the `list` function.

```
>>> id = [1, 2, 3, 4]
>>> leaders = ['Elon Musk','Tim Cook','Bill Gates','Yang Zhou']
>>> gender = ['small','small','male','male']
>>> record = zip(id, leaders)
>>> list(record)
[(1,'Elon Musk'),(2,'Tim Cook'),(3,'Bill Gates'),(4,'Yang Zhou')]
```

Calling the `zip` function for only one set will create as many single-element tuples as there are elements in the `id` sequence.

```
>>> record = zip(id)
>>> list(record)
[(1,), (2,), (3,), (4,)]
```

Using the no-argument `zip` function results in an empty set, and consequently using the `list` function produces an empty sequence.

```
>>> record = zip()
>>> list(record)
[]
```

The `zip` function can be used for more than two arguments. Here we present its call for three data sets: `id`, `leaders` and `gender`. The result is three-element tuples.

```
>>> id = [1, 2, 3, 4]
>>> leaders = ['Elon Musk','Tim Cook','Bill Gates','Yang Zhou']
>>> gender = ['small','small','male','male']
>>> record = zip(id, leaders, gender)
>>> list(record)
[(1, 'Elon Musk', 'small'), (2, 'Tim Cook', 'small'),
(3, 'Bill Gates', 'small'), (4, 'Yang Zhou', 'small')]
```

Using the `zip` function for unpacking. Using `*` - an asterisk against the `record` object resulted in unpacking the tuples from the list as four separate tuples:
`(1,'Elon Musk') (2,'Tim Cook') (3,'Bill Gates') (4,'Yang Zhou')`,
and then the `zip` function combines them into separate tuples of identifiers `id` and leaders `leaders`.

```
>>> record = [(1,'Elon Musk'), (2,'Tim Cook'), (3,'Bill Gates'),\
         (4,'Yang Zhou')]
>>> id, leaders = zip(*record)
>>> id
(1, 2, 3, 4)
>>> leaders
('Elon Musk', 'Tim Cook', 'Bill Gates', 'Yang Zhou')
```

```
>>> A = list(zip(*record))
>>> And
[(1, 2, 3, 4), ('Elon Musk', 'Tim Cook', 'Bill Gates', 'Yang Zhou')]
```

Now we will show how easily you can obtain the transposed matrix using the `zip` function.

```
>>> matrix = [[1, 2, 3], [1, 2, 3]]
>>>  matrix_T = [list(i) for i in zip(*matrix)]
>>> matrix_T
[[1, 1], [2, 2], [3, 3]]
```

List comprehensions are concise and a common code pattern for building result lists in Python. Depending on the Python version and the program code itself, list comprehensions can run much faster than hand-written `for` loop statements (often twice as fast). List comprehension iterations are performed within the interpreter at C speed, not Python speed. For this reason, using them can be beneficial from a performance perspective.

We can also use list comprehensions for file operations. If we open a file in an expression, the list comprehension will automatically read one line from the file at a time and add it to the results list. By using list comprehensions, we achieve a more efficient and faster solution. List comprehensions automatically close the file when the temporary object is cleaned up after the expression is executed.

```
>>> lines = [line for line in open('poem.txt')]
>>> lines
['Eeny, meeny, miny, moe,\n', 'Catch a tiger by the toe.\n',
'If he hollers, let him go,\n', 'Eeny, meeny, miny, moe.\n']
```

A `for` loop nested within a list comprehension expression can have an associated `if` clause, allowing you to "filter out" those result items for which the test is not true.

```
>>> file = open('poem.txt')
>>> lines = [line.rstrip() for line in file if line[0] == 'E']
>>> lines
['Eeny, meeny, miny, moe,', 'Eeny, meeny, miny, moe.']
```

8.2. Anonymous functions

In addition to the `def` statement, Python also provides a form of expression generating function objects called a `lambda expression` (or simply `lambda` for short). The name `lambda` is borrowed from `LISP`, which in turn borrowed it from lambda calculus (Alonzo Church 1930s; Church–Turing thesis), a form of symbolic logic. In Python, it's just a keyword that introduces the proper syntax for an expression. Like `def`, a lambda expression creates a function that can be called later, but it returns that function instead of assigning it to a name. For this reason, lambda expressions are sometimes called `anonymous` ("unnamed") functions. In practice, they are often used as a shorthand method, saving a function definition or delaying the execution of a fragment code.

The general form of a lambda expression consists of the keyword `lambda`, followed by any number of arguments, and then, after the colon, the expression:

```
lambda arg1, arg2, ..., argN : expression
```

Function objects returned by executing lambda expressions work exactly the same as those created and assigned by `def` instruction. Lambda expressions differ in that:

- `lambda` is not a statement but an expression, so it can appear in places where the usage of `def` is not allowed in Python syntax, e.g. inside a list literal or in function call;
- the body of `lambda` is a single expression, not a block of statements, and because of that it is used to write simple functions.

Here are some sample lambda expressions:

- takes two parameters `x` and `y` and returns their sum;

  ```
  lambda x, y: x + y
  ```

- takes one parameter and returns the value of that parameter increased by 1;

  ```
  lambda x : x + 1
  ```

- as above, but instead of the parameter name, we specify the underscore character `_`;

  ```
  lambda _ : _ + 1
  ```

- we assign the lambda expression to a variable and execute it;

  ```
  variable = lambda x,y: x+y
  variable(2,3)
  ```

- the invocation of the lambda expression itself.

  ```
  (lambda x,y: x+y)(3,4)
  ```

Listing 8.4 shows a program that uses the user-defined function `year` as a comparison key for searching and sorting.

Listing 8.4: *Using a function that returns a key for comparison*

```
def main():
  d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
  print('max:', max(d.items()))
  print('max by year:', max(d.items(), key=year))
  print('sorted:', sorted(d.items()))
```

```
6   print('sorted by year:', sorted(d.items(), key=year))
7 def year(item):
8   return item[1]
9
10 if __name__ == '__main__':
11   main()
```

After executing the program from listing 8.4, the following will appear on the screen: the maximum element found by the dictionary keys (line 3), the maximum element, but is searched by the dictionary values, i.e. the year (line 4), a list of tuples of dictionary elements sorted by its keys (line 5) and a list of tuples of dictionary elements sorted by its value, i.e. the year (line 6).

```
OUTPUT:
max: ('Eve', 1999)
max by year: ('Bob', 2003)
sorted: [('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
sorted by year: [('Eve', 1999), ('Cay', 2000), ('Ann', 2001),
('Bob', 2003)]
```

Recall that the `sorted` function creates a list of tuples corresponding to dictionary elements, sorted by the comparison key, which by default is the `<` operator applied to dictionary keys.

We can use a lambda expression as a comparison key in a sorting function (`sorted`) or to find the maximum element (`max`), e.g. the elements of a dictionary, as shown in listing 8.5.

Listing 8.5: *Using a lambda expression returning a comparison key*

```
d = {"Eve": 1999, "Ann": 2001, "Cay": 2000, "Bob": 2003}
print('max:',max(d.items()))
print('max by year:',max(d.items(), key=lambda t: t[1]))
print('sorted:',sorted(d.items()))
print('sorted by year:',sorted(d.items(), key=lambda t: t[1]))
```

The results of executing the program from listing 8.5 confirm the equivalence of the solutions used.

```
OUTPUT:
max: ('Eve', 1999)
max by year: ('Bob', 2003)
sorted: [('Ann', 2001), ('Bob', 2003), ('Cay', 2000), ('Eve', 1999)]
sorted by year: [('Eve', 1999), ('Cay', 2000), ('Ann', 2001),
('Bob', 2003)]
```

And some more examples of using lambda expressions in sorting functions:

- with a comparison key specified by a lambda expression referencing each key in the dictionary `d`:

```
>>> d = {'b': 42, 'c': 1, 'a': 2}
>>> sorted(d.items())
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0])
[('a', 2), ('b', 42), ('c', 1)]
>>> sorted(d.items(), key=lambda t: t[0], reverse=True)
[('c', 1), ('b', 42), ('a', 2)]
```

- with a comparison key specified by a lambda expression referencing each value in the dictionary `d`:

```
>>> d = {'b': 42, 'c': 1, 'a': 2}
>>> sorted(d.items(), key=lambda t: t[1])
[('c', 1), ('a', 2), ('b', 42)]
>>> sorted(d.items(), key=lambda t: t[1], reverse=True)
[('b', 42), ('a', 2), ('c', 1)]
>>> ob = sorted(d.items(), key=lambda t: t[1], reverse=True)
```

In `lambda` expressions, it is allowed to use a conditional expression of the form:

```
lambda arg1, arg2, ..., argN:
  value if true_expression [else [...]]
```

Depending on the value of `x`, an appropriate message is sent to the screen.

```
kom1 = "I will come today"
kom2 = "I won't come today"
print((lambda x: kom1 if x > 0 and x < 10 else kom2)(2))
# I will come today
print((lambda x: kom1 if x > 0 and x < 10 else kom2)(12))
# I won't come today
```

A higher-order function is an ordinary function with the difference that it takes another function as an argument or returns a function, e.g.

```
def function(f, number):
  return f(number)
```

The first argument is the function `f`, the second is `number`. A higher order function call using the function that calculates the square root of the number 2 looks like this:

```
import math
print(function(math.sqrt, 2))
```

Just for the purpose of calling a higher-order function, you can define an ordinary function:

```
def function(f, number):
  return f(number)
def cube(x):
  return x * x * x
print(function(cube, 2))
```

In turn, using a lambda expression, this notation can be shortened:

```
print(function(lambda x: x * x * x, 2))
```

8.3. Enumerated types

An `enumeration` is a set of symbolic names associated with unique, constant values. Within the enumeration, the elements belonging to it can be compared according to identity, and the calculation itself can be repeated. Since enumerations are used to represent constants, it is recommended to use names written in CAPITAL LETTERS for enumeration elements. In Python, the `enum` module defines four enumeration classes that can be used to define unique sets of names and values:

- `Enum` - base class for creating computed constants;
- `IntEnum` - base class for creating enumerated constants, which are also subclasses of `int`;
- `Flag` - a base class for creating computed constants that can be combined using bitwise operators without losing their affiliation to the `Flag` class;
- `IntFlag` - base class for creating computed constants that can be combined using bitwise operators without losing their affiliation with `IntFlag`; members of a class inheriting from `IntFlag` are also instances of `int`.

8.3.1. Class Enum

Enumerations are created using class syntax. To define an enumeration, create a subclass of `Enum` as follows:

```
from enum import Enum
class Color(Enum):
  RED = 1
  GREEN = 2
  BLUE = 3
```

Enumeration elements have a readable string representation, e.g.:

```
>>> Color.RED
<Color.RED: 1>
```

The type of an enumeration element is the enumeration to which the element belongs:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
```

Enumeration items have an attribute that contains only the name of the item:

```
>>> Color.RED.name
'RED'
```

Enumerations support iteration, in the order defined, e.g.

```
>>> for _ in Color:
...     print(_)
...
Color.RED
Color.GREEN
Color.BLUE
```

Sometimes it is useful to gain programmatic access to enum elements, e.g. in a situation where it is not possible to use `Color.RED` because the exact color is not known at the time of writing the program.

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

To access enum elements by name, you must use element access, e.g.

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If we have an element `enum` and we need its name or values we can refer to its appropriate attributes, e.g.

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

Having two enum elements with the same name is incorrect, which is reported as `TypeError`, e.g.

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last): ...
TypeError: Attempted to reuse key: 'SQUARE'
```

Two `enum` elements can have the same value. If two elements `A` and `B` have the same value, with `A` defined first, then `B` is an alias of `A`. Searching for a value corresponding to `A` and `B` will return `A`. Searching for the name `B` will also return `A`:

```
>>> from enum import Enum
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

In the module `enum`, there is a defined function, which is a decorator for enum classes (`@unique`). It searches the `__members__` attribute of the given enumeration and searches for all aliases it finds. If any are found, it will be reported. `ValueError` exception with details.

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last): ...
ValueError:
duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

If the exact value is not important, you can use the object class `enum.auto`, which, when used for the first time, will return a default value of 1 and, on each subsequent reference, will return a value 1, greater than the previous one.

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are selected by the function `_generate_next_value_`, which can be overwritten and customized to user requirements, e.g. by assigning names instead of natural numbers as values to created enumeration elements.

```
>>> from enum import Enum, auto
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     RED = auto()
...     GREEN = auto()
...
>>> list(Ordinal)
[<Ordinal.RED: 'RED'>, <Ordinal.GREEN: 'GREEN'>]
```

Iterating over the elements of an enumeration does not provide aliases, e.g. for a previously defined enumeration named `Shape` using a for loop will not show aliases.

```
>>> for element in Shape:
...     print(repr(item))
...
<Shape.SQUARE: 2>
<Shape.DIAMOND: 1>
<Shape.CIRCLE: 3>
```

The special attribute `__members__` is an ordered, read-only mapping of names to elements. It contains all the names defined in the enumeration, including aliases.

```
>>> for name, member in Shape.__members__.items():
...     print("({},{})".format(repr(name), repr(member)))
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The `__members__` attribute can be used to specify the details, programmatically accessing the elements of an enumeration, e.g. to find all aliases of the enumeration `Shape`.

```
>>> L = [n for n, m in Shape.__members__.items() if m.name != n]
>>> L
['ALIAS_FOR_SQUARE']
```

The elements of the enumeration are compared according to the identity:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordinal comparisons between enumeration values are not supported because the enumeration elements are not integers.

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError:'<' not supported between instances of 'Color' and 'Color'
```

However, equality comparisons are defined.

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons with non-computed values will always return `False`:

```
>>> Color.BLUE == 2
False
```

The `Enum` class is callable and provides API (*Application Programming Interface*), which resembles the semantics of the `namedtuple` class from the `collections` module. The first argument to the `Enum` function call is the name enum; the second argument is the source of the element names calculations. This can be a string of names separated by whitespace, a sequence of names, a sequence of 2-tuples with `key-value` pairs, or mapping (e.g. dictionary) of names to values. The last two options allow you to assign arbitrary values to enumerations. The rest automatically assign increasing integers, starting from 1. To specify a different starting value, use the start argument.

```
>>> Animal = Enum("Animal", "ANT BEE CAT DOG")
>>> Animal
<enum 'Animal'>
>>> repr(Animal.ANT )
'<Animal.ANT: 1>'
>>> Animal.ANT.value
1
>>> for animal in Animal:
...     print(repr(animal))
...
<Animal.ANT: 1>
<Animal.BEE: 2>
<Animal.CAT: 3>
<Animal.DOG: 4>
```

A new class derived from `Enum` is returned. In other words, the previous assignment to the variable `Animal` is equivalent to the following:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to 1 as the starting number and not 0, is that 0 has the value `False`, while all elements enumerations only accept the value `True`.

8.3.2. IntEnum class

Elements of the `IntEnum` class, which is also a subclass of `Enum` and `int`, can be compared with numbers integers. Furthermore, elements of various subclasses of the `IntEnum` class can be compared with each other.

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST=1;GET=2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, it is still not possible to compare the `Shape` subclass of `IntEnum` to the standard `Enum` enums.

```
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

`IntEnum` values behave like integers in a way that what to expect:

- casting to an integer type of an enumeration element returns the value of the `value` attribute of that element;

  ```
  >>> int(Shape.CIRCLE)
  1
  ```

- can be components of expressions;

  ```
  >>> Shape.CIRCLE + 5
  6
  ```

- can be indices of a sequence, such as a list;

  ```
  >>> ["a", "b", "c"][Shape.CIRCLE]
  'b'
  ```

- can be arguments to the `range` function.

  ```
  >>> [j for j in range(Shape.SQUARE)]
  [0, 1]
  ```

Objects of class `Shape` are both objects of class `int` and class `IntEnum`.

```
>>> issubclass(Shape, int)
True
>>> issubclass(Shape, IntEnum)
True
>>> isinstance(Shape.CIRCLE, int)
True
>>> isinstance(Shape.CIRCLE, IntEnum)
True
```

8.3.3. Flag Class

The `Flag` class is a subclass of the `Enum` class. `Flag` elements can be combined using the bitwise operators (`&`, `|`, `^`, `~`), but they cannot be combined either compare with any other `Flag` or `int` enum. Although it is possible to determine the value directly, it is recommended using the `auto`() value and letting the `Flag` class choose the appropriate value.

If the combination of elements does not set any flags, the value the logical result is `False`.

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
```

```
...     GREEN = auto()
...
>>> repr( Color.RED & Color.GREEN )
'<Color.0:0>'
>>> bool( Color.RED & Color.GREEN )
False
```

The individual flags should have values that are powers of two (i.e. `1`, `2`, `4`, `8`, `...`), while flag combinations will not be.

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()                #0001
...     BLUE = auto()               #0010
...     GREEN = auto()              #0100
...     WHITE = RED | BLUE | GREEN  #0111
...
>>> repr(Color.RED)
'<Color.RED: 1>'
>>> repr(Color.BLUE)
'<Color.BLUE: 2>'
>>> repr( Color.GREEN )
'<Color.GREEN: 4>'
>>> repr(Color.WHITE)
'<Color.WHITE: 7>'
```

Creating an element in an enumeration with value 0, i.e. "no flags set", means that its logical value is `False`.

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> repr(Color.BLACK)
'<Color.BLACK: 0>'
>>> bool(Color.BLACK)
False
```

8.3.4. Class IntFlag

Elements of the class `IntFlag`, which is a subclass of `Enum` and `int`, can be combined using the bitwise operators (`&`, `|`, `^`, `~`), and the result is still an element `IntFlag`. Furthermore, `IntFlag` elements can be used wherever `int` elements are used. The result of any operation on elements of class `IntFlag`, except bitwise operations, causes the loss of the belonging of this result to class `IntFlag`.

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> repr(Perm.R | Perm.W)
'<Perm.R|W: 6>'
>>> Perm.R + Perm.W
6
>>> type(Perm.R + Perm.W)
<class 'int'>
>>> RW = Perm.R | Perm.W
>>> type(RW)
<enum 'Perm'>
>>> repr(RW)
'<Perm.R|W: 6>'
>>> Perm.R in RW
True
>>> Perm.X in RW
False
```

It is also possible to give names to combinations of symbolic constants.

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     In = 2
...     X = 1
...     RWX = 7
...
>>> repr( Perm.RWX )
'<Perm.RWX: 7>'
```

```
>>> repr(~Perm.RWX)
'<Perm.-8: -8>'
```

Another important difference between `IntFlag` and `Enum` is that if no flags are set (value 0), then the logical value is `False`.

```
>>> repr( Perm.R & Perm.X )
'<Perm.0: 0>'
>>> bool(Perm.R & Perm.X)
False
```

Since instances of the `IntFlag` class are also instances of `int` class, you can combine one with the other.

```
>>> repr(Perm.X | 8 )
'<Perm.8|X: 9>'
```

The use of the `Enum` and `Flag` classes is recommended because `IntEnum` and `IntFlag` break some semantic promises for enumerations that are supposed to represent symbolic constants, by being comparable to integers, and thus by being transitive to other unrelated enumerations. In contrast, `IntEnum` and `IntFlag` should only be used in cases where `Enum` and `Flag` do not work, e.g. when integer constants are replaced by enumerations or for interoperability with other systems.

8.4. Iterators

An iterator is an object that allows sequential access to all the elements or parts contained in another object, usually a collection or a string. An iterator can be understood as a kind of pointer that provides two basic operations: referencing a specific element in the collection (element access) and modifying the iterator itself so that it points to the next element (sequential traversal of elements). There must also be a way to create an iterator that points to to the first element and how to determine when the iterator exhausted all items in the collection.

Depending on the language and intended use, iterators they may provide additional operations or have different additional behaviors. The primary purpose of an iterator is to allow the user process every item in a collection without having to delve into its internal structure. This allows the collection to store items in any way you want, while the user can treat it as a regular sequence or list. The iterator class is usually designed along with the class corresponding collection and is closely related to it. Typically, a collection provides methods to create iterators.

Iterators are one of the fundamental building blocks of Python, and are often completely unnoticeable because they are implicitly used in `for` loops. All the standard Python sequence types, as well many classes in the standard library, provide iteration. Iterators can also be defined explicitly. To get an iterator from a sequential collection, it uses the built-in function `iter()`. The built-in function `next()` returns on each call next element of the collection and modifies the iterator so that it pointed to the next item in the collection. When there are no more elements, this function throws a `StopIteration` exception. Example:

```
for x in ['a', 'b', 'c']:
  print(x)
```

Execution of the above loop involves:

- `Start of iteration`: from `obj`, which here is a list, an iterator is requested; `obj` gives it, by convention we denote it `it`.
- `First turn of the loop`:
  1. from `it` an object is requested;
  2. `it` gives the first element of `obj`, which is `'a'`;
  3. `'a'` is called `x`, the loop body (`print(x)`) is executed.
- `Second turn of the loop`: steps `1-3` are repeated, but this time the item given by `it` is `'b'`.
- `Third turn of the loop`: as above, for item `'c'`.
- The loop, wanting to start the fourth rotation, requests an object from `it`.
- `it` can no longer supply an object, reports end of iteration, the loop ends.

Each iteration has its own iterator, as the following example illustrates. For the list `lst`:

```
>>> lst = ['a', 'b', 'c']
```

We define the iterator `it1`:

```
>>> it1 = iter(lst)
```

And then we request from the iterator `it1` to give an element from the object `lst` and move to the next element:

```
>>> next(it1)
'and'
```

Now we define another iterator `it2`:

```
>>> it2 = iter(lst)
```

We check whether the previously created operators are not one and the same:

```
>>> it is it2
False
```

We find out that they are not, so we ask each of them to provide subsequent elements of the sequence, and the messages on the screen confirm the independence of the iterators.

```
>>> next(it2)
'and'
>>> next(it1)
'b'
>>> next(it1)
'c'
>>> next(it2)
'b'
```

Knowing that a `StopIteration` exception is thrown when the `next` method reaches the end of the sequence, we can use the `try` exception handling to prevent the program from terminating unexpectedly. To illustrate how it iterates through the elements of the `sequence` chain, we use the `while` loop and exception handling in the code in Listing 8.6.

Listing 8.6: *Using an iterator and handling the thrown exception* `StopIteration`

```
sequence = "Alice has a cat!!!"

it = iter(sequence)
try:
  while True:
    val = next(it)
    print(val)
except StopIteration:
  print("END")
```

Any user-defined class can provide standard, iteration (implicit or explicit) if it has a method `__iter__()` defined in its body that returns an iterator. The returned iterator must also have methods `__iter__()` and `__next__()` defined.

In Python, iterables represent sequences of objects that can be iterated over: examples include `for` loops that work on iterables, constructs lists or foldable dictionaries. Many methods or functions from the standard libraries (and not only) are written so that their arguments can be any objects iterable, e.g. `sorted(iterable)`. In Python, for an object to be recognized as an iterable, it and its iterators must implement the so-called `iterator protocol`:

- The iterable object `ob` must implement the special method `__iter__()`, returning an iterator.
- The iterator must implement the methods:
  - `__next__()`, which either returns an object, or throws an exception representing the end of iteration (`StopIteration`);
  - `__iter__()`, returning an iterator (it can be – and usually it is – himself).

Starting with Python 3.4, the most accurate way to check if object is iterable, is to call the function `iter()` and handle the `TypeError` exception if it is not. Examples of iterable objects:

- strings (objects of class `str`),
- byte sequences (objects of classes `bytes` and `bytearray`),
- lists (objects of class `list`),
- sets (objects of class `set`),
- tuples (objects of class `tuple`),
- dictionaries (objects of class `dict`),
- files (objects returned by the `open` function),
- ranges (objects of class `range`).

Notice that the objects in all the above examples are iterable objects, but they are not iterators.

The examination of iterator types of selected iterable objects can be performed as follows:

```
for type in (str, tuple, list, set, dict):
  ob = type()    # new object of given type
  it = iter(ob)    # new iterator of this object
  print(type(it))

print(type(iter(range(0)))) # range() invalid
```

```
OUTPUT:
<class 'str_iterator'>
<class 'tuple_iterator'>
<class 'list_iterator'>
<class 'set_iterator'>
<class 'dict_keyiterator'>
<class 'range_iterator'>
```

The name of the dictionary iterator `dict_keyiterator` suggests that the iterator is used to iterate over the keys of a dictionary. And so it is in reality:

```
>>> d = {'a': 1, 'b': 2, 'c': 42}
```

```
>>> for kind:
...  print(k)
...
and
b
c
```

Dictionaries have methods that return iterable objects representing values and `key-value` pairs:

```
>>> values = d.values()
>>> print(type(values), type(iter(values)))
<class 'dict_values'> <class 'dict_valueiterator'>
>>> items = d.items()
>>> print(type(items), type(iter(items)))
<class 'dict_items'> <class 'dict_itemiterator'>
>>> # Below we iterate over d.items(), returned
>>> # objects are "consumed" by the list constructor
>>> print(list(d.items()))
[('a', 1), ('b', 2), ('c', 42)]
```

Objects that are iterators in Python follow the protocol iterative, which basically means they provide two methods: `__iter__()` and `__next__()`. An iterator is an object representing a stream of data; this object returns one item at a time from the data stream. The Python iterator must support a method called `__next__()`, which takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, the method `__next__()` must throw a `StopIteration` exception. Iterators do not have to be finite; it is reasonable to write an iterator that generates an infinite stream of data.

In listings 8.7 and 8.8, we present, respectively, the definition of the infinite iterator class `PowersOfTwo` generating successive powers of two and a program using this iterator to print successive powers of 2 on the screen (lines 11-14) no greater than the given upper limit (line 7 or 9). We draw attention to the need to define in the main program the condition for terminating the infinite iterator.

LISTING 8.7: *Class of iterator returning squares of consecutive natural numbers*

```
1 #powersoft.py
2 class PowersOfTwo:
3   def __init__(self):
4     self.num = 1
5   def __iter__(self):
```

```
    return self
  def __next__(self):
    num = self.num
    self.num *= 2
    return number
```

LISTING 8.8: *Using the* `PowersOfTwo` *class iterator*

```
# main_powersoftwo.py
from sys import argv
from powersoftwo import PowersOfTwo

def main():
  try:
    limit = int(argv[1])
  except:
    limit = 10000
  it = iter(PowersOfTwo())
  a = next(it)
  while a < limit:
    print(a)
    a = next(it)

if __name__ == "__main__":
  main()
```

One of the most common actions performed on lists and other iterable objects is to apply some operation to each of their elements and collect the results. Because this is such a common operation, Python provides an appropriate built-in function that is able to do it for us.

- The `map` function is used to apply the passed function to each element of the iterable object and returns the iterator object containing all the results of its call:

  ```
  map(function, *iterables) -> map object
  ```

  The returned iterator object can be converted to a list using the function built-in `list`.

  We will use the `map` function to create a list of squares of numbers between 0 and 5, where for each value in that range we will apply a lambda expression to calculate its square.

  ```
  >>> list(map(lambda x: x * x, range(6)))
  [0, 1, 4, 9, 16, 25]
  ```

Special attention should be paid to the number of iterable objects in the `map` function call, which must be equal to the number of arguments to the function called in it, e.g. the standard `pow` function requires two arguments: the base and the exponent. The mapping causes the following operations to be performed: `0**0` (in the case of Python, no exception is raised for this unmarked symbol, it is simply assumed that the result is 1), `1**1`, ..., `5**5`.

```
>>> powers = map(pow, range(6), range(6))
>>> list(powers)
[1, 1, 4, 27, 256, 3125]
```

The `map` function terminates after the shortest iterable object has been exhausted.

```
>>> powers = map(pow, range(6), range(9))
>>> list(powers)
[1, 1, 4, 27, 256, 3125]
```

- The `filter` function filters out elements of an iterable based on a testing function:

```
filter(function, iterable) -> filter object
```

Elements of the iterable object for which the testing function returns `True` are added to the results list. The `filter` function returns an iterator object containing the filtered elements. This object can be converted to a list using the function built-in `list`.

As an example, we will show how to use the `filter` function to select only odd numbers from various iterable objects.

```
>>> numbers = [x for x in range(10)]
>>>
>>> def odd(x): return x % 2 == 1
>>>
>>> odd = filter(odd, numbers)
>>> print(list(odd))
[1, 3, 5, 7, 9]
>>>
>>> odd = filter(odd, range(10))
>>> print(list(odd))
[1, 3, 5, 7, 9]
>>>
>>> odd = filter(lambda x: x % 2 == 1, range(10))
>>> print(list(odd))
[1, 3, 5, 7, 9]
```

- Calling the function `reduce(function,iterable)` from the *functools* module, where the first argument `function` is a binary function and the second `iterable` is iterable object, returns a single value calculated as follows:
  - function `function` takes the first two elements of an object `iterable` i computes the result;
  - function `function` takes the previous result and the third item `iterable` object and computes the result;

    ...
  - function `function` gets the previous result and the last item from the `iterable` object and calculates the result.

  The operation of the `reduce` function can be simulated by defining the following function:

```
def reduce(function, iterable, initializer=None):
  it = iter(iterable)
  if initializer is None:
    value = next(it)
  else:
    value = initializer
  for element in it:
    value = function(value, element)
  return value
```

  We will show several examples of using the `reduce` function for various iterable objects, performing basic arithmetic operations on them, and for strings, concatenating their characters along with reversing their order.

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4])
10
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4])
24
>>> reduce(lambda x, y: x * y, {1, 2, 3, 4})
24
>>> reduce(lambda x, y: x + y, range(1,5))
10
>>> s = "Kiler sentenced to good changes"
>>> reduce(lambda x, y: y + x, s)
'ynaimz erbod an ynazaks reliK'
>>> reduce(lambda x, y: x + y, s)
'Kiler sentenced to good changes'
```

8.4.1. Module itertools

The `itertools` module (module documentation is available at the link: https://docs.python.org/3/library/itertools.html) implements a series of iterators inspired by constructs from `APL`, `Haskell` and `SML` languages. The module standardizes a basic set of fast, efficient memory tools that are useful on their own or in connection. Together, they form an "iterator algebra" that allows specialized tools to be constructed concisely and efficiently in pure Python.

For example, the `SML` language provides the `tabulate(f)` tool, which creates the sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining `map()` and `count()` functions, creating `map(f, count())`.

The `itertools` module tools and their built-in counterparts work well they also work with quick functions in the `operator` module. For example, the multiplication operator can be mapped to two vectors, creating an efficient dot product:
`sum(map(operator.mul, vector1, vector2))`.

```
>>> import operator
>>> a = [1,2,3]
>>> b = [3,4,5]
>>> sum(map(operator.mul, a, b))
26
```

The descriptions of the functions from the `itertools` module, which we present below, have been enriched with examples of their operation performed in the interpreter. To correctly perform each of the examples, after starting the interpreter, you must import the necessary modules:

```
>>> from itertools import *
>> import operator
```

- Function `count(start=0, step=1)` creates an iterator that returns values starting with the one given as the `start` parameter. Subsequent values are incremented by the value of `step`, which is the second parameter.

  ```
  count(10) --> 10 11 12 13 14 ...
  count(2.5, 0.5) --> 2.5 3.0 3.5 ...
  count(2, -3) --> 2 -1 -4 -7 -10 -13 ...
  ```

  We now introduce a lower bound for the values returned by `count`, exceeding which causes the loop to terminate.

  ```
  >>> for num in count(2,-3):
  ...     print(num, end=", ")
  ```

```
...     if num < -20:
...         break
...
2, -1, -4, -7, -10, -13, -16, -19, -22,
```

- Function `cycle(iterable)` takes as a parameter an `iterable` object, through which this iterator passes. However, if the object runs out (reaches the end), it will still return values from the beginning. This is because `cycle` saves each element in memory.

```
>>> c = cycle("ABCD")
>>> for i in range(10):
...     print(next(c), end=", ")
...
A, B, C, D, A, B, C, D, A, B,
```

- Function `repeat(object, times=None)` returns the given object `times` times. If the `times` parameter is not passed, the object will be returned infinitely. The official documentation explains that `repeat` is mainly used as a constant argument passed to the `map` function. It can also be used to add a constant value to a tuple.

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Function `accumulate(iterable, func = operator.add)` creates an iterator that returns cumulative sums or cumulative results of other binary functions, specified with the optional argument `func`. The elements of the iterable object `iterable` can be of any type of the type that the `func` function allows. For example, for default addition operation elements can be any types. If the argument `iterable` has no elements, then the result also will have no elements.

```
>>> date = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
```

- Function `chain(*iterables)` returns an object of class `chain` whose method `__next__` returns elements from the first iterable object, up to its exhaustion, and then the elements from the next object iterable until all objects are exhausted.
  `chain.from_iterable(iterable)` - an alternative constructor of class `chain` that takes a single iterable argument.

```
>>> list(chain("Ala", "ma", "cat"))
['A', 'l', 'a', 'm', 'a', 'k', 'o', 't', 'a']
>>> data = ["Ala", "has", "cat"]
>>> list(chain.from_iterable(data))
['A', 'l', 'a', 'm', 'a', 'k', 'o', 't', 'a']
```

- Function `compress(iterable, selector)` creates an iterator that filters elements from an iterable object `iterable`, returning only those that have a corresponding element in selector `selector` evaluates to `True`. Stops when either the `iterable` object or the object is exhausted `selector`.

```
>>> list(compress("ABCDEF", [1, 0, 1, 0, 1, 1]))
['A', 'C', 'E', 'F']
>>> list(compress(count(2, 3), repeat(True, 8)))
[2, 5, 8, 11, 14, 17, 20, 23]
>>> list(compress(cycle("ABC"), repeat(True, 8)))
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B']
```

- Function `dropwhile(predicate, iterable)` creates an iterator that removes elements from an iterable object so as long as the predicate is true and then returns each element. This iterator does not produce any output until the predicate will not become false, so its startup time may be long.

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8]
>>> list(dropwhile(lambda x: x < 8, L))
[8, 1, 2, 3, 4, 5, 6, 7, 8]
>>> L = [x for x in range(50000001)]
>>> list(dropwhile(lambda x: x < 50000000, L))
[50000000]
```

- The `filterfalse(predicate, iterable)` function creates an iterator that filters elements from the `iterable` object, returning only those for which `predicate` is false. If predicate is equal to `None`, returns elements that are false.

```
>>> list(filterfalse(lambda x: True, range(10)))
[]
>>> list(filterfalse(lambda x: False, range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(filterfalse(lambda x: None, range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(filterfalse(lambda x: x % 2, range(10)))
[0, 2, 4, 6, 8]
```

```
>>> list(filterfalse(bool, range(10)))
[0]
>>> list(filterfalse(None, range(10)))
[0]
```

- The function `groupby(iterable, key=None)` creates an iterator that returns successive keys and groups from `iterable`. The argument `key` is a function that computes a value for each element. If the argument `key` is not specified or is set to `None`, then this argument becomes the identity function by default. Basically, the `iterable` argument should already be sorted by the `key` function.

```
>>> L = [k for k, g in groupby('AAAABBBCCDAABBB')]
>>> L
['A', 'B', 'C', 'D', 'A', 'B']
>>> L = [list(g) for k, g in groupby('AAAABBBCCDAABBB')]
>>> L
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D'],
['A', 'A'], ['B', 'B', 'B']]
```

- Function:
`islice(iterable, start, stop[, step])`
`islice(iterable, stop)`
creates an iterator for retrieving data slices from an object iterable.

```
>> "".join(islice('ABCDEFG', 2))
'AB'
>>> "".join(islice('ABCDEFG', 2, 4))
'CD'
>>> "".join(islice('ABCDEFG', 2, None))
'CDEFG'
>>> "".join(islice('ABCDEFG', 0, None, 2))
'ACEG'
>>> list(islice(count(0), 10, 30, 2))
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

- The `starmap(function, iterable)` function creates an iterator that returns the results of applying the function `function` to arguments obtained from an iterable object, whose elements are tuples of length equal to the number function arguments.

```
>>> list(starmap(operator.add,[(2,3),(4,5),(1,1)]))
[5, 9, 2]
```

- Function `takewhile(predicate, iterable)` creates an iterator that returns those elements from the iteration for which the predicate is true.

```
>>> list(takewhile(lambda x: x < 5,[1,4,6,4,1]))
[1, 4]
>>> list(takewhile(lambda x: x < 9, count(3)))
[3, 4, 5, 6, 7, 8]
```

- The `tee(iterable, n = 2)` function returns a tuple of `n` independent iterators from a single iterable object.

```
>>> list(t[0])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> list(t[1])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> list(t[2])
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

- The `zip_longest(*iterables, fillvalue=None)` function creates an iterator that aggregates the elements from each iterable. If the number of iterations is of uneven length, the missing values are filled with the fill value. The iteration continues until the longest iteration is exhausted.

```
>>> list(zip_longest("ABCD", "xy", fillvalue="-"))
[('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

- The `product(*iterables, repeat=1)` function creates the Cartesian product of iterable objects.

```
>>> for elem in product("ABCD", "xy"):
...     print(elem)
...
('A', 'x')
('A', 'y')
('B', 'x')
('B', 'y')
('C', 'x')
('C', 'y')
('D', 'x')
('D', 'y')
>>> for elem in product("ABCD", "xy", repeat = 2):
...     print(elem)
```

```
...
('A', 'x', 'A', 'x')
('A', 'x', 'A', 'y')
('A', 'x', 'B', 'x')
('A', 'x', 'B', 'y')
('A', 'x', 'C', 'x')
('A', 'x', 'C', 'y')
('A', 'x', 'D', 'x')
('A', 'x', 'D', 'y')
('A', 'y', 'A', 'x')
('A', 'y', 'A', 'y')
('A', 'y', 'B', 'x')
('A', 'y', 'B', 'y')
('A', 'y', 'C', 'x')
('A', 'y', 'C', 'y')
('A', 'y', 'D', 'x')
('A', 'y', 'D', 'y')
... # itd.
>>> for elem in product(range(2), repeat=3):
...     print(elem)
...
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
```

- The function `permutations(iterable, r=None)` creates tuples of length `r`, of all possible orderings, with no repeated elements.

```
>>> for elem in permutations("ABC", 2):
...     print(elem)
...
('A', 'B')
('A', 'C')
('B', 'A')
('B', 'C')
```

```
('C', 'A')
('C', 'B')
>>> for elem in permutations(range(2)):
...     print(elem)
...
(0, 1)
(1, 0)
```

- Function `combinations(iterable, r)` creates substrings of length `r` from the elements of the object `iterable`.

```
>>> for elem in combinations("ABCD", 2):
...     print(elem)
...
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'C')
('B', 'D')
('C', 'D')
>>> list(combinations(range(4), 3))
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
```

- The `combinations_with_replacement(iterable, r)` function creates substrings of length `r` from the elements of the `iterable` object, allowing individual elements to be repeated more than once.

```
>>> for elem in combinations_with_replacement("ABC", 2):
...     print(elem)
...
('A', 'A')
('A', 'B')
('A', 'C')
('B', 'B')
('B', 'C')
('C', 'C')
```

8.5. Generators

Generators are a special case of iterators that, thanks to the iteration concept, allows you to write programs with better performance. There are two types of generators:

- `Generator functions`: These are similar to regular functions, except that instead of returning a result via the `return` statement, they use a special `yield` statement that allows them to suspend and resume their state between calls.
- `Generator expressions`: These are similar to list comprehensions, except instead of returning a list, they return an object that produces the results in order. The syntax is the same as for list comprehensions, except that parentheses are used instead of square brackets.

The `yield` keyword is used similarly to `return`. Executing the `return` instruction permanently transfers control to the place where it was called. Executing the `yield` instruction, on the other hand, transfers control, but only temporarily, because it puts the function in which it was called to sleep and remembers its local state. Each function that contains the `yield` instruction is a function generator, and every generator is an iterator.

When the generator function is called, the arguments are assigned to the local function context (as in normal functions), but the code inside the function is not executed. The generator function returns a generator object that is iterator. Each time the `next()` function is called on this object, the generator function code is executed until it encounters `yield` or `return` statements or to the end of the function. When the encountered instruction is `yield`, the state of the generator function is frozen, and the value on the right side of `yield` is returned to the code executing the `next()` function. However, if the generator exits in a way other than using the `yield` instruction, it permanently stops generating new values. Therefore, there is no need to raise the `StopIteration` exception, which is certainly a significant convenience. Since the object returned by the generator function is an iterator, such a function can be used in a simple way, e.g. in the `for` loop, as shown in listing 8.9. The generator function `gen_squares` generates successive squares of non-negative integers one after another.

LISTING 8.9: *Example of a simple generator function*

```python
def main():
  gen = gen_squares(6)
  print(type(gen))
  for num in gen:
    print(num, end=" ")
def gen_squares(n):
  for j in range(n):
    yield j*j
if __name__ == '__main__':
  main()
```

If we did not use a generator function, we would have to immediately build a list of all obtained values in the function, which, in the case of a large number of results, causes memory consumption and takes a lot of time. Generators allow you not only to save memory but also to evenly distribute the generation time data while allowing other code to process its generated partial data. For more advanced applications, generators can be an alternative to the manual saving state between iterations in class objects. In the case of generators, variables available in the function scopes are automatically saved and restored.

Listing 8.10: *Example of using generator function as infinite iterator*

```
def gen_squares(start):
  while True:
    yield start * start
    start += 1

g = gen_squares(10)
print(type(g))
for j in gen_squares(10):
  print(j, end=" ")
  if j > 300:
    break
```

Generators can be created by short `generator expressions`, which have a form similar to foldable lists, but instead of creating a list, they create a generator that "lazily" returns the next objects. For example, we can list the first eight squares construct as a comprehensible list:

```
>>> a = [num * num for num in range(8)]
>>> and
[0, 1, 4, 9, 16, 25, 36, 49]
```

Or define a generator expression that does not store all the values but is an iterator thanks to the `next` function and returns one value at a time.

```
>>> g = (num * num for num in range(8))
>>> g
<generator object <genexpr> at 0x7f47bcb8b220>
>>> next(g)
0
```

Generator expressions are an alternative to other objects iterables or containers that require more space in memory, such as lists, tuples, or sets. To illustrate the difference, we use the `getsizeof` function from the `sys` module, which returns the size of the object in bytes, and the `randrange` function from the `random` module, which must be imported before executing the following instructions. For a list comprehension, we get `89_095_160B`:

```
>>> a = [randrange(1,1000) for j in range(10000000)]
>>> sys.getsizeof(a)
89095160
```

Whereas for the generator expression it is only `104B`

```
>>> g = (randrange(1,1000) for j in range(10000000))
>>> sys.getsizeof(g)
104
```

Using the generator reduces the memory footprint by `856_684` times.

```
>>> sys.getsizeof(a)//sys.getsizeof(g)
856684
```

Python 3.3 introduced an extended statement syntax `yield`, which uses the `from generator` clause to delegate the action to a subgenerator. In simple cases, it is equivalent to the `for` loop, the use of which is shown in listing 8.11, where calling the `list` function forces generator to generate all values at once.

LISTING 8.11: *Example of using* `for` *loop in generator*

```
def main():
  print(list(both(6)))
def both(n):
  for j in range(n): yield j
  for j in (x ** 2 for x in range(n)): yield j
if __name__ == '__main__':
  main()

#OUTPUT:
# [0, 1, 2, 3, 4, 5, 0, 1, 4, 9, 16, 25]
```

Using the `from generator` clause makes the code more concise and readable and supports all common usage contexts generator, as illustrated in listing 8.12.

LISTING 8.12: *Example of using the* `from` *clause in the generator*

```
def main():
  print(list(both(6)))
  print(" : ".join(str(j) for j in both(6)))

```

```
def both(n):
  yield from range(n)
  yield from (x ** 2 for x in range(n))

if __name__ == '__main__':
  main()

#OUTPUT:
# [0, 1, 2, 3, 4, 5, 0, 1, 4, 9, 16, 25]
# 0 : 1 : 2 : 3 : 4 : 5 : 0 : 1 : 4 : 9 : 16 : 25
```

Consider the examples of a generator expression `generator_1` and a generator function `generator_2`.

```
>>> generator_1 = (x for x in [1, 2, 3, 4, 5])
>>> def generator_2():
...     yield 1
...     yield 2
...     yield 3
...     yield 4
...     yield 5
...
```

We will use the module `collections.abc` and the functions `Iterator`, `Iterable` and `Generator` to show the interdependencies of object types.

```
>>> from collections.abc import Iterator, Iterable, Generator
>>> type(generator_1)
<class 'generator'>
>>> type(generator_2)
<class 'function'>
>>> type(generator_2())
<class 'generator'>
>>> isinstance(generator_1, Generator)
True
>>> isinstance(generator_2(), Generator)
True
>>> isinstance(generator_1, Iterator)
True
>>> isinstance(generator_2(), Iterator)
True
```

```
>>> isinstance(generator_1, Iterable)
True
>>> isinstance(generator_2(), Iterable)
True
>>> issubclass(Generator, Iterator)
True
```

An interesting feature of the generator is the ability to inject a value into it. The `yield` instruction not only returns a value but can also accept a value from somewhere else. To send "something" to the generator, simply execute the `send()` method.

In listing 8.13, we present an example of a generator that generates a new integer every `0.5` second, starting from one. The loop will define a condition that if the number 10 is generated, then the number `-1` will be sent to the generator (line 8). This is interpreted by the generator as a jump used to generate the next number, which will make the generator count from 1 to 10 and then loop back to end at one. The word `return` will be used to terminate the generator.

LISTING 8.13: *Example of using the* `send` *method*

```
1  from time import sleep
2
3  def main():
4    generator = generator_4()
5    for x in generator:
6      print(x)
7      if x == 10:
8        print(generator.send(-1))
9
10 def generator_4():
11   and = 1
12   step = 1
13   while True:
14     new_step = yield and
15     if new_step:
16       step = new_step
17     and += step
18     if i == 0:
19       return
20     sleep(0.5)
21
22 if __name__ == '__main__':
23   main()
```

Generators are a much easier way to create iterators, especially instead of those built from a class. In the vast majority of cases, this is possible. The most important advantage of iterators and generators is the saving of resources. It is recommended to pass generator expressions instead of list comprehensions to functions that expect iterable objects, such as `min()`, `max()`, and `sum()`, among others. This is not only more efficient but also fits into the Python "philosophy".

8.6. Practice exercises

8.6.1. Iterators

1. Write a program that, given a list of cities, prints these cities using an iterator and the iteration statement `while`. Use the exception `StopIteration`.
2. Write a program `biglist.py` that, for a number `n`, creates a list of numbers from `1` to `n`, and then calculates the sum of the numbers from this list. Call the program as follows: `/usr/bin/time -f "%e sec. %M kB" python biglist.py n` where `n` is a specific number.
3. Write a program `bigrange.py` that calculates the sum of the numbers from `1` to `n` for the number `n`. Do not create a list, but use the built-in function `range`. Call the program as follows: `/usr/bin/time -f "%e sec. %M kB" python bigrange.py n` where `n` is a specific number.
4. Using the information provided by the `/usr/bin/time` program for the two programs above, draw the appropriate conclusions.
5. Write a program that, given a list of Celsius temperatures, converts that list to a list of Fahrenheit temperatures. Use the expression `lambda` and the built-in function `map`.
6. Write a program that, given a list of temperatures in Fahrenheit, converts that list into a list of temperatures in Celsius. Use the expression `lambda` and the built-in function `map`.
7. Write a program that generates a list of Fibonacci numbers of a given length, and then uses the `lambda` expression and the built-in `filter` function to generate a list of odd numbers. Fibonacci numbers.
8. Write a program that generates a list of Fibonacci numbers of a given length, and then uses a lambda expression and the built-in function `filter` to create a list of even numbers. Fibonacci numbers.
9. Write a program that, for a given list of numbers, finds the largest number using the built-in function `max` and the function `reduce` from the module `functools`.

10. Write a program that finds the largest number for a given list of numbers using the appropriate lambda expression and the `reduce` function from the `functools` module. In the function call `reduce`, do not use the built-in `max` function.
11. Write a program that, given a list `list1` of one hundred random numbers from the interval `<-10,10>`, transforms this list into a list `list2` of their squares. Use the expression `lambda` and the built-in function `map`. Then calculate and print to the screen the sum of the elements of the list `list2` using `iterator` and the iteration instruction `while` and using the exception `StopIteration`.

8.6.2. Itertools module

1. Define the following functions using functions from the `itertools` module and place them in a file called `iterutils.py`:
   (a) `get(n,iterable)` - returns a list of the n initial elements of the iterable object `iterable`. When `n > len(iterable)` returns a list of all elements of `iterable`.
   (b) `append(value,iterator)` - returns an iterator whose first returned value is `value` and the next ones are values returned by iterator `iterator`.
   (c) `table(func,start=0)` - returns an iterator whose subsequent values are: `func(start)`, `func(start + 1)` etc.
   (d) `nth(iterable,n,default=None)` - returns the `n`-th element from the `iterable` object or the value of `default`.
   (e) `such_same(iterable)` - returns `True` if all elements are equal.
   (f) `count(iterable, pred=bool)` - returns the number of elements of the iterable object `iterable` for which the predicate `pred` is true.
   (g) `fill(iterable)` - returns an iterator providing subsequent elements of the iterable object `iterable`, and then value of `None` indefinitely.
   (h) `ntimes(iterable, n)` - returns an iterator that repeats the elements of the iterable object `n` times.
   ```
   list(ntimes("ab", 3) -> ["a", "b", "a", "b", "a", "b"]
   ```
   (i) `flat(list_of_lists)` - returns an iterator that flattens one level of nesting.
   ```
   list(flatten([[1, 2, 3],[5, 6]])) -> [1, 2, 3, 4, 5]
   ```
   (j) `repeat(func,times=None,*args)` - returns an iterator that repeats the calls to `func` with the specified arguments.
   ```
   repeat(random.random)
   list(repeat(math.sin,3,math.pi/2)) -> [1.0, 1.0, 1.0]
   list(repeat(math.pow,4,2,3)) -> [8.0, 8.0, 8.0, 8.0]
   ```

2. Write a program `test_iterutils.py`, and in it, in the function `main`, test the functions from the file `iterutils.py`.

8.6.3. Generator functions

1. Write a program with the generator function `divisible_by_3_and_5(limit)`. This function should provide consecutive natural numbers from the interval `[0, limit)` that are divisible by `3` and by `5`. In the `main` function, load a natural number into the n variable and print all the numbers returned by function: `divisible_by_3_and_5` called with argument n.
2. Write a program with a no-argument generator function with the header: `divisible_by_3_and_5`. This function should provide the consecutive natural numbers divisible by `3` and by `5`. In the `main` function, load a natural number into the variable `n` and print all the numbers supplied by the function `divisible_by_3_and_5`, which are less than `n`.
3. Write a program with a generator function `fibonacci(limit)`. This function should provide successive Fibonacci numbers from the interval `[0,limit)`. In the `main` function, read a natural number into the variable `n` and print all the numbers supplied by function `fibonacci` called with argument `n`.
4. Write a program with a generator function `fibonacci`. This function should provide the next numbers Fibonacci. In the `main` function, read a natural number into the variable `n` and print all numbers provided by the `fibonacci` function that are less than `n`.

8.6.4. Generator expressions

1. Write a program that uses a generator expression to create a list of integers from the range `[1..n]` divisible by `3` or divisible by `5`, where `n` is the number given as the first argument program.
2. Write a program that uses a generator expression to create a `n`-element list (`numbers`) of numbers of type `float` from the range `[-100..100]` rounded to two decimal places (use the `random.uniform` function). Then print the list `numbers` to the screen and use generator expressions for each of the following points, write one instruction that prints:
   a) a list of positive numbers from list `numbers`;
   b) a list of positive numbers from list `numbers` converted to type `int`;
   c) a list of values of the function `math.floor` for the numbers from the list `numbers`;
   d) a list of values of the function `math.ceil` for the numbers from the list `numbers`;

e) a list of values of the function `math.log` for positive numbers from the list `numbers`.

8.6.5. Additional tasks

1. Write a program that:
   a) using the function `generate(n)`, generates a sequence of numbers from `1` to `n`;
   b) uses a generating expression to create a sequence of numbers from `1` to `n`.

   Then in the `main` function, it will print every second generated number for each of the subpoints separately:
   1) using `iterator` and the iteration statement `while` and using the exception `StopIteration`;
   2) using a `for` loop.
2. Write a program that uses an infinite generator with header: `multiple(n)`, to find the next multiples of the number `n` passed as an argument. Then print 10 such numbers separated by commas:
   a) using `iterator` and the iteration statement `while` and using the exception `StopIteration`;
   b) using a `for` loop.

Bibliography

[1] Beazley David, Brian K. Jones, *Python Cookbook*, 3rd Edition, O'Reilly Media Inc., USA, 2013.

[2] Dawson Michael, *Python programming for the Absolute Beginner*, 3rd Edition, Course Technology PTR, USA, 2010.

[3] Downey Allen B., *Think Python: How to Think Like a Computer Scientist*, 2nd Edition, O'Reilly Media Inc., USA, 2016.

[4] Lutz Mark, *Python Pocket Reference*, Fifth Edition, O'Reilly Media Inc., USA, 2014.

[5] Lutz Mark, *Learning Python*, 5th Edition, O'Reilly Media, Inc. USA, 2020.

[6] Matthes Eric, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, 3rd Edition, No Starch Press Inc., USA, 2023.

[7] Ramalho Luciano, *Fluent Python: Clear, Concise, and Effective Programming Devices*, 2nd Edition, O'Reilly Media, Inc. USA, 2022.

[8] Slatkin Brett, *Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series)*, Pearson Education Inc., 2015.

[9] Summerfield Mark, *Programming in Python 3: A Complete Introduction to the Python Language (Developer's Library)*, 2nd Edition, Addison-Wesley Professional, 2009.

[10] Wentworth Peter, Elkner Jeffrey, Downey Allen B., Meyers Chris, *How to Think Like a Computer Scientist: Learning with Python 3*, `http://openbookproject.net/thinkcs/python/english3e/`

[11] `https://www.python.org`

[12] `https://www.python.org/doc/`

The Publishing House of Jan Dlugosz University in Czestochowa
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl