Python - lecture notes

with example programs from archive ***python_ examples***

Tomasz R. Werner

Warsaw, May 28, 2024 22:21

Contents

Page

1 Intro . . . 1
- 1.1 History . . . 1
- 1.2 General remarks on the language . . . 1
- 1.3 Python shell . . . 2
- 1.4 The *IDLE* IDE . . . 5
- 1.5 The Zen of Python . . . 5
- 1.6 A few remarks about the style . . . 6
- 1.7 Built-in types . . . 7
  - 1.7.1 Numeric types . . . 7
  - 1.7.2 Sequence types . . . 7
  - 1.7.3 Sets and dictionaries . . . 9
  - 1.7.4 Other types . . . 10
- 1.8 Objects and references . . . 10
- 1.9 Functions . . . 14
- 1.10 Modules . . . 16
- 1.11 Help . . . 17
- 1.12 Keywords (reserved words) . . . 20

2 Flow control statements . . . 21
- 2.1 *pass* statement . . . 21
- 2.2 *if* statement . . . 22
- 2.3 *if-else* (conditional) expression . . . 22
- 2.4 *while* loop . . . 23
- 2.5 *for* loop . . . 24
- 2.6 *match* statement . . . 27

3 Basic data types . . . 31
- 3.1 *NoneType* type . . . 31
- 3.2 *ellipsis* type . . . 32
- 3.3 Typ *NotImplementedType* . . . 32
- 3.4 Numeric types . . . 32
  - 3.4.1 Integers . . . 33
  - 3.4.2 Boolean type . . . 35
  - 3.4.3 Floating point numbers . . . 36
  - 3.4.4 Complex numbers . . . 37
- 3.5 Strings (introduction) . . . 38

4 Operators . . . 40
- 4.1 Assignment (not an)operator . . . 40
- 4.2 Assignment (true) operator . . . 41
- 4.3 Operators and operator precedence . . . 42
- 4.4 Parenthesized expressions . . . 42
- 4.5 Subscription and slicing . . . 43
  - 4.5.1 Indexing . . . 43
  - 4.5.2 Slices . . . 45
- 4.6 Call operator . . . 48
- 4.7 await expression . . . 49
- 4.8 Exponentiation . . . 49
- 4.9 Unary operators . . . 50
- 4.10 Arithmetic operators . . . 50

4.10.1 Division operators . . . 51
4.10.2 Modulo operator . . . 52
4.11 Bitwise operators . . . 53
4.12 Comparisons . . . 57
4.12.1 Arithmetic comparisons . . . 57
4.12.2 Identity comparisons . . . 58
4.12.3 Membership test operators . . . 58
4.13 Boolean *not*, *and* and *or* operators . . . 59
4.14 Conditional expression . . . 60
5 Functions . . . 63
5.1 Defining functions. Scopes of variables . . . 63
5.2 Passing arguments . . . 69
5.3 Lambdas . . . 74
5.4 Some “special” functions. . . . 78
5.4.1 *filter* . . . 79
5.4.2 *map* . . . 80
5.4.3 *reduce* . . . 80
5.4.4 *accumulate* . . . 82
5.4.5 *sorted* . . . 83
5.5 Function decorators. . . . 85
6 Comprehensions and generators . . . 93
6.1 Comprehensions . . . 93
6.1.1 List comprehension . . . 94
6.1.2 Set comprehension . . . 97
6.1.3 Dictionary comprehension . . . 98
6.2 Generator functions . . . 99
6.2.1 Generator comprehension . . . 102
6.3 Communicating with generators . . . 105
6.4 Pipes of generators and coroutines . . . 109
6.5 Some tools from *itertools* . . . 113
7 Collections . . . 118
7.1 Sequential collections . . . 118
7.1.1 Non-modifying operations . . . 118
7.1.2 Modifying operations . . . 121
7.1.3 *bytes* objects . . . 125
7.1.4 *bytearray* objects . . . 127
7.2 Hashable objects . . . 128
7.3 Sets . . . 130
7.4 Dictionaries . . . 137
7.5 Other collections . . . 142
7.5.1 Factory function *namedtuple* . . . 143
7.5.2 Class *defaultdict* . . . 145
7.5.3 Class *Counter* . . . 146
7.5.4 Class *deque* . . . 150
7.6 Unpacking iterables . . . 153
8 Strings . . . 158
9 Regular expressions . . . 165
9.1 Basic concepts . . . 165
9.1.1 Metacharacters . . . 165
9.1.2 Classes . . . 166

9.1.3 Predefined classes of characters . . . 166
9.1.4 Special locations . . . 167
9.1.5 Quantifiers . . . 168
9.1.6 Substitutes for methods using regexes of class *String* in Java . . . 168
9.2 Pattern objects . . . 170
9.3 Capturing groups . . . 171
9.3.1 Plain groups — (...) . . . 171
9.3.2 Non-capturing groups — (?: ...) . . . 174
9.3.3 Named groups — (?P<name> ...) . . . 174
9.3.4 Internal backreferences . . . 176
9.3.5 External backreferences . . . 177
9.3.6 Lookahead and lookbehind assertions — (?= ...), (?! ...), (?<= ...), (?<! ...) 178
9.3.7 Yes-no patterns . . . 180
9.3.8 Comments — (?# ...) . . . 181
9.4 Option flags . . . 181
9.4.1 *MULTILINE* . . . 183
9.4.2 *DOTALL* . . . 183
9.4.3 *IGNORECASE* . . . 184
9.4.4 *ASCII* . . . 184
9.4.5 *VERBOSE* . . . 185
9.5 Functions operating on regular expressions . . . 186
9.5.1 re.compile . . . 186
9.5.2 re.search . . . 186
9.5.3 re.match . . . 187
9.5.4 re.fullmatch . . . 188
9.5.5 re.split . . . 188
9.5.6 re.findall . . . 189
9.5.7 re.finditer . . . 190
9.5.8 re.sub . . . 190
9.5.9 re.subn . . . 191
9.5.10 re.escape . . . 191
9.5.11 re.purge . . . 192
9.6 Matcher methods . . . 192
9.6.1 matcher.groups, matcher.group . . . 193
9.6.2 matcher.groupdict . . . 194
9.6.3 matcher.start, matcher.end, matcher.span . . . 194
9.6.4 matcher.lastindex, matcher.lastgroup, matcher.re, matcher.string . . . 194
9.7 Pattern object attributes . . . 195
10 Exceptions . . . 197
10.1 Raising and handling exceptions . . . 197
10.2 Built-in and custom exceptions . . . 207
10.3 Assertions . . . 209
11 Classes . . . 211
11.1 Introduction . . . 211
11.2 Constructing objects . . . 216
11.3 Instance and static fields . . . 220
11.4 Instance, static and class methods . . . 222
11.4.1 Non-static methods . . . 222
11.4.2 Class methods . . . 223
11.4.3 Static methods . . . 225

11.5 Duck typing 227
11.6 Properties 227
11.7 String representation of objects 231
11.8 Magic methods 234
11.8.1 Operator overloading 234
11.8.2 Other magic methods 242
12 List of listings 246
Index 248

Chapter 1

Intro

1.1 History

Python was initially developed by Guido van Rossum[1] in the late 1980's in the Netherlands. At that time, he worked in a project, called Amoeba, at the CWI (Centrum voor Wiskunde en Informatica) in Amsterdam. The aim of the project was to build a distributed operating system. As a part of it, a general purpose programming language called ABC was developed by a team that van Rossum belonged to. He was not quite satisfied with the language, so in December 1989 he started working on a new language, based on ABC but without some of its problematic features.

The first version of Python (version 0.9.0) was published in February 1991. It already had classes with inheritance, exceptions, functions and fundamental data types (numbers, lists, dictionaries, strings). It was based on the concept of modules (inspired by Modula-3).[2]

Guido van Rossum

In 1994, discussion forum ***comp.lang.python*** was created, accelerating the growth of the language and its popularity.

Python 2 was released on 16th of October 2000 and its last version is 2.7. It will not be continued and has been replaced by Python 3, first released on 3rd of December 2008. Python 3 contains many new features making it *not* fully compatible with Python 2 (it contains the ***2to3*** utility which automates the translation of Python 2 code to Python 3).

Guido van Rossum played the rôle of the lead developer of Python (as the "benevolent dictator for life") until 12 July 2018, when he announced his "permanent vacation" from his responsibilities, which were taken over by a five-member Steering Council. Development of the language is now lead by the non-profit organization Python Software Foundation.

The name of the Python language comes from a BBC television comedy-sketch series *Monty Python's Flying Circus*.

1.2 General remarks on the language

Python is an object-oriented language: everything, including integers and floats, is an object. Even classes and modules are implemented as objects!

Python is a **strongly typed** language, which means that every object has a definite type. At the same time, it is **dynamically typed**, meaning that there is no type-checking of the code prior to running it (Java and C/C++ are *statically* typed; types of all objects have to be known to the compiler and cannot be changed).

Python is an interpreted language. Technically, the code *is* compiled into a form of byte-code before it's executed. However, this process of compilation is "invisible" from

[1] For correct pronunciation of his name, see https://www.youtube.com/watch?v=-pz5vhTdkpA

[2] A language, published in 1988, which was never adopted for industrial use, but heavily influenced Java, Python, C# and other modern languages.

the user's point of view, who sees the code immediately executing without a compilation phase.

Python is a general-purpose programming language, not intended for use in any particular domain or environment. There are, of course, some areas where it's less suitable than other languages, mainly because it's quite time and memory consuming (this can be remedied by leveraging Python modules written in C).

It is a multi-paradigm programming language supporting object-oriented programming, structured programming, many features of functional programming (inspired by Lisp, Haskell and ML).

It has a garbage collector, as Java, and, also as Java, a huge standard library in the form of packages and modules that can be imported to user's programs. What's more, there is an official collection of third-party Python software, called ***PyPI***.[1] A special application (***pip***) can be used to easily install these packages. Currently (February 2023), ***PyPI*** contains more than 440 000 packages! They can be used for just about anything: automation, data analysis, managing databases, documentation, building graphical user interfaces, image and sound processing, machine learning, multimedia, networking, scientific computing, system administration, test frameworks, web frameworks and so on...

1.3 Python shell

The best way to start programming in Python is to use the Python shell called REPL — **R**ead what the user types in, **E**valuate what was entered, **P**rint the value of the expression entered, **L**oop back and wait for the next input. It can be launched just by entering ***python*** (on some systems ***python3***) and then entering Python instructions at the prompt (which is `>>>`). If the instruction is an expression, i.e., it has a value (different than **None**), then this value will be printed after pressing `ENTER`:

```
$ python
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 7
>>> y = 12
>>> x+y
19
>>> [a**2 for a in range(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> exit()
```

By the way: as this example demonstrates, assignment (here `x=7` and `y=12`) is *not* an expression in Python[2] — that's why entering `x=7` does not print any result (see sec. 4.1, p. 40 and 4.2, p. 41).

To exit the REPL, use **exit()**, or Ctrl-D (Linux/Mac), or Ctrl-Z-Enter (Windows).

When typing instructions in REPL (but *not* in Python scripts!), you can use a special "variable" called _ (underscore) — it refers to the last value evaluated and printed:

[1] Python Package Index: `https://pypi.org/`

[2] In C/C++/Java, it has the type and value of the (reference to) the variable on the left-hand side after assignment.

```
>>> a = 6
>>> a
6
>>> b = _ + 3   # _ is 6
>>> b
9
>>> _ * 8       # _ is 9
72
```

In order to enter a piece of data from the keyboard, you can use – in both REPL and scripts – the **input** function, optionally passing as argument a prompt to be used:

```
>>> whatever = input("Enter whatever: ")
Enter whatever: Hello
>>> whatever
'Hello'
>>> noprompt = input()
there was no prompt!
>>> noprompt
'there was no prompt!'
```

The function returns whatever was typed as a string; if what you need is a number, you have to convert this string to **int** or **float** (the latter corresponds to **double** in C/C++/Java):

```
1     >>> i = input("Enter an int: ")
2     Enter an int: 123
3     >>> i, type(i)
4     ('123', <class 'str'>)
5     >>> i = int(i)
6     >>> i, type(i)
7     (123, <class 'int'>)
8     >>>
9     >>> f = input("Enter a float: ")
10    Enter a float: 123.45
11    >>> f, type(f)
12    ('123.45', <class 'str'>)
13    >>> f = float(f)
14    >>> f, type(f)
15    (123.45, <class 'float'>)
16    >>> print(i, f)
17    123 123.45
18    >>> print("Numbers are " + str(i) + " and " + str(f))
19    Numbers are 123 and 123.45
```

Some explanations:

- Line 3: two (or more) comma-separated values create a tuple (see sec. 7.1, p. 118) which is then printed (line 4) as parenthesized sequence of these values.
- Line 3: the **type** function (actually, a constructor of class **type**) returns an object representing the type (class) of the argument — what is printed is its string representation.

- Lines 5 and 13: functions **int** and **float** (actually, constructors of the corresponding classes) return numbers represented by strings passed as argument.
- Lines 5 and 13: note that we assign the results of function calls to the same variables (i and f, respectively). This variables (references) now refer to objects of different types: they referred to objects of type **str** (string), and now they refer to **int** and **float**, respectively (compare lines 4 with 7 and 12 with 15).

In lines 16 and 18, we used the **print** function. It accepts any number of comma-separated arguments and prints their values converted to strings using a space as the separator. This can be changed by supplying the sep argument:

```
>>> a = 7
>>> b = 1.5
>>> c = 'Hello'
>>> print(a, b, c)
7 1.5 Hello
>>> print(a, b, c, sep=' - ')
7 - 1.5 - Hello
```

By default, **print** (like **println** in Java) adds new-line character at the end. It also may be changed by supplying the end argument (it can be an empty string, and then nothing will be added):

```
a = 123
b = 123.45
c = 'Hello'
print(a, b, sep=' and ', end=' and then ')
print(c)
```

prints

```
123 and 123.45 and then Hello
```

Note that a and b are numbers, not strings, but they were converted to strings automatically.

As in Java, you *can* concatenate strings using the + operator, but both operands must then be strings. In Java, assuming k is an integer, something like `"k = " + k` is correct: k will be automatically converted to string and concatenated with "k = ". To do the same in Python, first you have to convert k to string for the concatenation to work:

```
>>> k = 3
>>> print("k = " + str(k))
k = 3
>>> print("k = " + k)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Python uses (and its interpreter enforces) a little strange convention of writing the code: blocks (of loops, ifs, function and class definitions, ...) are marked by indentation and by indentation *only* (no braces, begin...end and such)! The amount of indentation is in principle arbitrary, but the convention is to use always 4 spaces

— do *not* use TABs![1] This also applies to REPL. For example, the following loop has a body consisting of two lines. Note that after beginning of the loop,[2] the REPL expects its body, so the prompt changes from `>>>` to `...`. Both lines of the body have to be indented by the same amount of spaces (4 is recommended); after the last line of a block, just enter a blank line in order to revert to the indentation level used before the loop

```
>>> for n in [2, 6, 8, 10]:
...     m = 2*n + 1
...     print(m)
...
5
13
17
21
```

1.4 The IDLE IDE

You can also use an IDE which usually comes with standard Python installations: it is called ***IDLE*** (Python's Integrated Development and Learning Environment) and can be launched just by typing ***idle*** in the console. It is a simple, lightweight application but for writing small programs and for experimenting may be quite sufficient. It has the following features:

- coded in 100% pure Python, using the standard ***tkinter*** package to create graphical user interface;
- cross-platform: works mostly the same on Windows and Linux/MacOS;
- embedded Python shell window (interactive interpreter) with colorizing of code input, output, and error messages;
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features;
- search within any window, replace within editor windows, and search through multiple files (***grep***);
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces;
- configuration, browsers, and other dialogs.

For details, see the manual[3].

1.5 The Zen of Python

Every Pythonista[4] knows the so called Zen of Python — a collection of 20 "guiding principles" for writing programs in Python (published by Tim Peters in 1999) only 19 of which are known. Zen of Python can be found as PEP 20[5] or viewed by typing `import this` in the Python shell (REPL):

[1] All Python-oriented editors will automatically insert 4 spaces when you press TAB at the beginning of a line.

[2] Instructions requiring a body — like loops or **if**s — end with a colon.

[3] `https://docs.python.org/3/library/idle.html`

[4] Someone who uses (and is a fan of) the Python programming language.

[5] `http://peps.python.org/pep-0020`

```
>>> import this
The Zen of Python, by Tim Peters

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

By the way, PEPs[1] (Python Enhancement Proposals) are documents describing new features proposed for Python and also various aspects of Python, like its design and style.

1.6 A few remarks about the style

Python is a high-level language and hence the Python programs are just text files. It can be written in many ways and as long at it is syntactically correct the interpreter does not care. However, it is important, for the author and for all who will ever read the program, to use a consistent style which will make the code easy to read and understand. A common style facilitates collaboration on projects and makes it easier to change them later. The famous PEP 8[2] describes in details the recommended (but not enforced) rules of writing clear code in Python (it is continuously updated as the language evolves). Every Python developer should read the whole document; below, we just mention some of these rules:

- Lines shouldn't be longer that 79 characters.
- Continuation lines should be indented by four extra spaces.
- Function and class definitions should be separated by two blank lines.
- Methods in a class should be separated by one blank line.
- Don't put spaces around indices, function calls, or keyword argument assignments.
- Spaces should be used around operators and after commas.
- Comments (from # to the end of line) should be on a line of their own.
- A docstring (which will be discussed later) should be defined for all non-trivial functions, classes and modules.

[1] https://peps.python.org

[2] http://peps.python.org/pep-0008

- It is recommended to use CamelCase (starting with a capital letter) for the names of classes and exceptions.
- Use lower_case_with_underscores for the names of functions, variables, and attributes; the camelCase is also acceptable (but starting with a lowercase letter).
- Names of variables which are meant to be constant are written in ALL_CAPITALS with underscores.
- The first parameter of (non-static) methods, which is the reference to the object the method is invoked on, should be named self.[1]
- The first parameter of class methods should be named cls.
- Imports should be in sections in the following order: standard library modules, third-party modules, your own modules. Imports in each subsection should be in alphabetical order.

1.7 Built-in types

We will describe Python types in a separate section (see sec. 3, p. 31); here we will just briefly enumerate some of them without too many details.

1.7.1 Numeric types

There are only three built-in numeric types: **int**, **float** and **complex**. They can be freely mixed in arithmetic operation — even integers have well defined real *and imaginary* parts! Objects of numeric types are immutable.

Type **int** represents integer values with practically unlimited values, negative or positive.

Type **float** describes real numbers (as a matter of fact, an "infinitely tiny" subset of rational numbers, because of finite representation on 64 bits). They are implemented as **doubles** (*not* **floats**!) known from C/C++/Java languages.

Type **complex** describes complex numbers represented as pairs of **floats** — their real and imaginary parts.

1.7.2 Sequence types

There are built-in types objects or which represent sequences of objects (strictly speaking, references to objects). These are **list**, **tuple** and **range**. Sequences and collections in Python are really sequences or collections of *references* to objects, as are collections in Java. However, references in Python are *not* just bare pointers, as they are in Java (more about it later).

Type **list** corresponds somehow to **ArrayList** from Java or **vector** in C++. It's a list of references to objects of any types (also to other lists or tuples). It is denoted by comma-separated list enclosed in square brackets. Lists are mutable — we can add of remove elements or replace a reference to an object with a reference to another object without changing the reference to the list itself and leaving ID number the same:

```
mylist = []                # empty list
print(type(mylist))
print(id(mylist))
```

[1] Note that in C++/Java this parameter is *not* mentioned on the parameter list at all, so there is no way to give it a name and therefore its name is fixed once for all as this.

```
mylist.append(1)
mylist.append([11, 22])
mylist.extend([33, 44])
print(mylist)
print(id(mylist))
del mylist[3]
print(mylist)
print(id(mylist))
```

prints

```
<class 'list'>
140144996134208
[1, [11, 22], 33, 44]
140144996134208
[1, [11, 22], 33]
140144996134208
```

Type **tuple** represents an immutable sequence of references to objects of any type. Immutability means that you can neither add or remove elements of a tuple, nor replace a reference which is its element by another reference. However, if this is the reference to a mutable object (as a list), this object *can* be modified. Tuples are denoted by comma-separated lists enclosed in round parentheses (not always required) — in one-element tuple, its sole element must be followed by a comma:

```
mytuple = (5, 'Hello', [1, 2, 3], (4,))
print(type(mytuple))
print(type(mytuple[3]))
print(mytuple)
mytuple[2].pop(1)
print(mytuple)
mytuple[1] = 'World'
```

prints

```
<class 'tuple'>
<class 'tuple'>
(5, 'Hello', [1, 2, 3], (4,))
(5, 'Hello', [1, 3], (4,))
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py",
        line 578, in runcode
  exec(code, self.locals)
File "/home/werner/p.py", line 7, in <module>
  mytuple[1] = 'World'
TypeError: 'tuple' object does not support item assignment
```

Type **range** represents an immutable sequence of integers which is an arithmetic progression. It is mainly used in loops, but is used in numerous other situations. Constructor of **range** takes the value of the first element, the limit such that all elements must be smaller (or bigger for negative difference) than that, and the difference between consecutive elements (`step`). Constructor with one argument `n` is equivalent to three arguments: 0, `n` and 1. Note that ranges themselves are *not* collections: they just yield elements "on demand":

```
myrange1 = range(5, 10, 2)
print(myrange1)
print(list(myrange1))
print(list(range(24, -2, -5)))
print(list(range(5)))
```

prints

```
range(5, 10, 2)
[5, 7, 9]
[24, 19, 14, 9, 4, -1]
[0, 1, 2, 3, 4]
```

1.7.3 Sets and dictionaries

Type **set** represents sets, as in Java or C++. It is a mutable collection of references to objects of any type. The implementation is based on hashing,[1] so elements must be immutable (more precisely, hashable). There is also the **frozenset** type: it represents immutable (and hashable) set, so it can be used as an element of another set. For example:

```
myset = {1, 2, frozenset(['Hello', 'World']), 'Paris'}
print(type(myset))
print(2 in myset)
anotherset = {3, 4}
sum  = myset | anotherset          # sum
prod = myset & set([1, 2, 3, 4])   # product
print(myset-anotherset)            # difference
print(anotherset)
print(sum)
print(prod)
```

prints

```
<class 'set'>
True
{frozenset({'World', 'Hello'}), 1, 2, 'Paris'}
{3, 4}
{1, 2, 3, 'Paris', 4, frozenset({'World', 'Hello'})}
{1, 2}
```

Type **dict** corresponds to **LinkedHashMap** in Java (maps are called dictionaries in Python). It represents a set of key/value pairs, where keys must be unique, immutable and hashable. For example:

```
mymap = {}       # empty dictionary, not set
mymap['France'] = 'Paris'
mymap[2] = [1, 2, 3]
mymap[2].append(4)
print(2 in mymap)
anothermap = {'Spain' : 'Madrid', 2 : complex(1, 2)}
```

[1] As **HashSet** in Java or **unordered_set** in C++.

```
mergedmap = mymap | anothermap
print(mymap)
print(anothermap)
print(mergedmap)
```

prints

```
True
{'France': 'Paris', 2: [1, 2, 3, 4]}
{'Spain': 'Madrid', 2: (1+2j)}
{'France': 'Paris', 2: (1+2j), 'Spain': 'Madrid'}
```

Implementation of the Python language itself is based on dictionaries.

1.7.4 Other types

There are a few other built-in types that we will encounter later. The most important of them is, of course, **str** representing strings, but we will postpone details for special chapter.
Note, however, that there is no type corresponding to **char** from other languages — a character is just a one-letter string of type **str**.

1.8 Objects and references

Names of variables that we use in Python are in fact names of *references* to objects. These references are really C-structures[1] in memory holding pointers to objects representing the type of the current value and to structures representing this value, reference counter, two more pointers needed by the garbage collector, etc.
Structures representing values have also internal identifier, or ID number (an integer, usually just the memory address of the object) — it is guaranteed that **no two objects accessible at the same time may have the same ID.** During the execution of the program, the same reference (and thus name) may change its interpretation and refer to another value, of different type and ID. In other words, Python has strongly typed objects, but untyped variable names.

There is a built-in function, **id**, which, given a reference to an object, returns, as an integer, the ID number of the object referred to by this reference. In the following example, we also use another built-in function, **type**, which returns the object representing the type of the value pointed to by reference passed as the argument:

```
>>> x = 3
>>> y = 7
>>> id(x), id(y)
(9353376, 9353504)         # 1
>>> x = y                  # 2
>>> id(x), id(y)
(9353504, 9353504)         # 3
>>> y = 'Python'           # 4
>>> id(x), id(y)
(9353504, 139733635191952) # 5
>>> type(x), type(y)       # 6
(<class 'int'>, <class 'str'>)
```

[1] Not just “bare” pointers, as in Java.

As one can see (line 1), identities of objects referred to by references x and y are different: 9353376 and 9353504, respectively. After the assignment[1] x = y (line 2), both x and y refer to exactly the same object (line 3) of type **int** and with value 7. What happened to the object with identity 9353376? It is lost; it's not accessible any more and will be removed from memory by the garbage collector.
In line 4, we assign another value to y — now it's a string. As we can see from the output, the ID, type (and value) of x and y are now different.

Note that even literal values are treated as references to objects, so one can call a method on them; for example, the type (class) **int** has the **bit_length** method which returns number of bits needed to represent a given value, while strings (class **str**) have the **upper** method

```
>>> (1).bit_length()
1
>>> (255).bit_length()
8
>>> (256).bit_length()
9
>>> 'paris'.upper()
'PARIS'
```

(The parentheses around the **int** literals are necessary; otherwise the dot would be interpreted as a decimal dot.)

Programs in Python, like in other languages, can be written in any text editor. To run it, one can simply type

```
python3 python_program.py
```

or just

```
python python_program.py
```

depending on configuration.
A Python program can also be made directly executable by adding, on Linux/Mac, the so called 'shebang' as the first line and setting the mode of the file to 'x' (executable). It was also a good practice to specify explicitly, in the second line, the encoding of the source file. However, in Python 3, it is UTF-8 by default, so many programmers do not do it anymore:

Listing 1 AAB-First/First.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# a comment
v = 1.25                              # 1
print(id(v))
if 1 <= v <= 2:
    print('v^2 =', v**2, end = '; ')
    print('v^3 =', v**3, end = '; ')
```

[1] What we assign here are references, not values.

```
    print('v^4 =', v**4)
    v = "now I'm a string!"           # 2
    print(id(v))
print('This is v:', v)
```

The program prints

```
20164608
v^2 = 1.5625; v^3 = 1.953125; v^4 = 2.44140625
140529498173656
This is v: now I'm a string!
```

Note that the object pointed to by v initially (line 1) represents a **float** with identity 20164608, but after modification (line 2), the same reference v points to another object with different identity (140529498173656) and different type (**string**). The object with its ID equal to 20164608 is now lost.
Note also, that exponentiation is denoted, as in Fortran, by double asterisk (**) – never by ^!
In this example, not only the value, but also the type has been modified. However, objects of all numerical types are in fact *not* modifiable (they are *immutable*), so assigning a new value to a number actually creates a new object:

```
>>> x = 1
>>> id(x)
9353312
>>> x = 2
>>> id(x)
9353344
```

There are many immutable types in Python; beside numerical types, a very important example is the **str** type representing strings:

```
>>> x = 'John'
>>> y = x
>>> id(x), id(y)
(139733656059312, 139733656059312)
>>> x = x + ' and Mary'
>>> x, y
('John and Mary', 'John')
>>> id(x), id(y)
(139733654847152, 139733656059312)
```

As we can see, after modification, x represents now a new object with different ID.

On the other hand, objects of many other types *are* modifiable (they are *mutable*). An important example is **list**. A list is, well… a list of *references* to objects (of any type). We can modify a list, for example, by appending a new element. However, the object representing this list will remain the same and its ID will *not* be changed. This is illustrated by the following example:

Listing 2 AAC-Refs/Refs.py

```
 1 #!/usr/bin/env python3
 2 # -*- coding: utf-8 -*-
 3
 4 from copy import deepcopy
 5
 6   # a and be are two names of the same objects
 7 a = [1, 2, [3, 5]]
 8 b = a
 9 print(b is a)        # True
10 a[2][1] = 4
11 print(b, '\n')       # [1, 2, [3, 4]]
12
13   # shallow copy: lists distinct, elements are copied
14   # but these are references referring to the same objects!
15 c = list(a)
16 c.append(5)
17 print(c is a)        # False
18 print(c[2] is a[2]) # True
19 a[2][1] = 6
20 print(a)             # [1, 2, [3, 6]]
21 print(c, '\n')       # [1, 2, [3, 6], 5]
22
23   # deep copy: lists distinct, elements too (recursively)
24 d = deepcopy(a)
25 d.append(5)
26 print(d is a)        # False
27 print(d[2] is a[2]) # False
28 a[2][1] = 7
29 print(a)             # [1, 2, [3, 7]]
30 print(d)             # [1, 2, [3, 6], 5]
```

In line 7, we create a list a: its first two elements are references to (immutable) integers and the third is the reference to another list. After b = a, the two names, b and a, refer to the same list object. We make a change using the name a (line 10): a[2] is the third element of the list (indexing is zero-based) which is the reference to the (mutable) list [3, 5] and, consequently, a[2][1] is its second element (reference to integer 5). Then we print b: as the output shows, the element b[2][1] has been changed, as a and b denote in fact the same object.
This can be seen from the output of the line 9: the **is** operator compares identifiers (addresses) and yields **True** only when we compare exactly the same object (there is also == operator which compares *values* — see Ch. 4, p. 40).

In line 15, we create a new list c, passing to the constructor (more on lists and constructors later, see sec. 7.1, p. 118 and sec. 11.1, p. 211). Now c and a are different list objects (line 17). However, elements of c are copies of elements of a and these elements are in fact *references*, not values, so they point to the same objects; that's why print(c[2] is a[2]) prints **True**. This element is itself modifiable (as it's a list), so changing one of its element (line 19) affects both a and c. Such coupling of lists is often undesirable — in order to create independent copy of a list, one can use **deepcopy** function form the ***copy*** module (line 24).

1.9 Functions

Functions will be described in more detail later (see sec. 5, p. 63). Here, we will just show an example illustrating creating and using a simple function.

As everything else, functions in Python are represented by objects. Definition of a function looks as shown in the example below. After the **def** keyword, we specify the name of a reference that we want to assign the function to, then a list of parameters (without types!) in parentheses and the colon indicating that what follows will be a block of instructions which define the function. The block that follows has to be indented, because, as we know, there is no other way to indicate where a compound instruction begins and where it ends. As in other languages, if a function returns a nontrivial result, the last executed statement must be **return**. All functions in Python return something — if there is no **return**, the special value **None** will be returned.

Notice that in line 3 we define a function (an object representing it) and at the same time we bind the object representing the function to a name, **adder** in our case. In python, this is an *executable* statement, like assignment. The name **adder** is not the name of this function that has to be used to invoke it. Rather, it's the name of a *reference* to this function. If we want, we can copy this reference to another — we do it on line 17

Listing 3 AAY-Funcs/funcs.py

```
 1  #!/usr/bin/env python
 2  
 3  def adder(a, b):          # definition + assignment
 4      return a + b
 5  
 6  print(type(adder))        # type
 7  
 8  rn = adder(7,4)           # adding numbers
 9  print(rn, type(rn))
10  
11  rs = adder('Monte ', 'Carlo')  # 'adding' strings
12  print(rs, type(rs))
13  
14  print(adder.__str__())   # invoking method on a function!
15  
16  print(id(adder))
17  a = adder                 # copying reference, now a is adder
18  print (id(a))
19  print(a is adder)         # a and adder are the same thing
20  
21  adder = a(1.25, 3.75)     # now adder becomes a float
22  print('Now adder is', adder, type(adder))
23  
24  print("But 'a' is still our function!", a(2.5, 3.5))
25  
26  print('='*30 + '\nAnd now list of attributes:')
```

```
27 print(str(dir(a)) + '\n' + '='*30)
28
29 print('But attribute a.__name__ is still ' + a.__name__)
30 a.__name__ = 'whatever'
31 print('No problem, now it is ' + a.__name__)
```

and from now on references **a** and **adder** refer to exactly the same object and are equivalent. We can even change the type of the reference **adder**, so now it refers to a **float** (line 21), and we still have access to the function through the reference **a** (line 24).

The program prints

```
<class 'function'>
11 <class 'int'>
Monte Carlo <class 'str'>
<function adder at 0x7f078af0bd90>
139670372531600
139670372531600
True
Now adder is 5.0 <class 'float'>
But 'a' is still our function! 6.0
==============================
And now list of attributes:
['__annotations__', '__builtins__', '__call__', '__class__',
'__closure__', '__code__', '__defaults__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__init_subclass__',
'__kwdefaults__', '__le__', '__lt__', '__module__',
'__name__',  '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
==============================
But attribute a.__name__ is still adder
No problem, now it is whatever
```

For people raised on C/C++/Java, it may seem strange that nowhere in the definition of the function have the types of parameters and of the return value been specified. In Python this is not only not required, but actually there is no way to do it![1] This means that we can pass to **adder** any two values of arbitrary types, as long as the + operator used in its definition works![2]

Being objects, functions have attributes: actually, quite many of them (even *methods* – line 14). We can see their list by invoking the **dir** function, as on line 27. Among these attributes, there is also `__name__`, which is still `adder` (line 29). But this is just a string, and if it bothers us, we can replace it with whatever we wish (what we do on line 30).

[1] There are methods to indicate these types, but only for human readers or third-party applications; the interpreter will just ignore this information anyway.

[2] This is an example of the so called *duck typing*.

1.10 Modules

As in Java or C/C++, in order to use the standard library, we have to somehow "import" its relevant parts to our program.

In Java, we often use the **import** keyword to do it. As the matter of fact, **import** in Java is not necessary at all, it just makes the code easier to type and read. Something like:

```
import java.util.List;
```

essentially just tells the compiler *"whenever you encounter the name `List`, replace it by its full, qualified name `java.util.List`"*. If you are fine with typing `java.util.List` instead of just `List`, you don't need any imports.

In C/C++ we have **#include** preprocessor directive. This is different:

```
#include <map>
```

physically includes the contents of file ***map*** and sends it together with our code to the compiler.[1]

Python is an interpreted language, so there is no separate compilation phase; everything happens at runtime. In Python, we use **import** keyword, but it's more like **#include** from C/C++ than **import** from Java.

For example,

```
import math
```

imports (and *executes*) module (file) ***math***. Modules correspond to normal Python source files with extension *.py*. They are looked for in directories from the list of paths, which we can see like this[2]

```
>>> import sys
>>> sys.path
['', '/usr/lib/python38.zip', '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '/home/werner/.local/lib/python3.8/site-packages',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/lib/python3/dist-packages']
```

(empty string at the beginning means *current directory*.) When found, the file is read in by the Python interpreter *and executed*. Most often, modules contain only definitions of classes, functions, constants, etc., but if there are, e.g., **print** statements there, we will see their output.

The names of all entities defined in the module will be put into the namespace whose name is the same as the name of the module, so to access them, we will have to qualify them with the name of the namespace. For example, module ***math*** contains definition of **sqrt** function. To access it, we type **math.sqrt**

```
>>> import math
>>> math.sqrt(234.567)
15.315580302424065
```

[1] Compilers use clever tricks to make the process more efficient and to avoid recompiling the same code over and over again, but these are technicalities.

[2] We can add more directories to this list, see sec. **??**, p. **??**.

Sometimes it's more convenient (or is expected by tradition in a given community of the users) to give a module another name (alias) under which we will refer to it

```
>>> import math as m
>>> m.sqrt(234.567)
15.315580302424065
```

We can import only selected definitions from a module, and then their names don't have to be qualified:

```
>>> from math import sqrt, factorial
>>> sqrt(999.99)
31.6226184874055
>>> factorial(45)
119622220865480194561963161495657715064383733760000000000
```

If we want to save on typing long names, or to avoid name clashes, we can rename entities imported from a module:

```
>>> from math import sqrt as root, factorial as fac
>>> root(87)
9.327379053088816
>>> fac(12)
479001600
```

Finally, we can import all names from a module, although it's not recommended, because then it's sometimes not so easy to tell what comes from which module and may lead to name clashes:

```
>>> from math import *
>>> from time import *
>>> gcd(3219, 1508)
29
>>> factorial(23)
25852016738884976640000
>>> log(999, pi)
6.033523597161682
>>> 4*atan(1.0)
3.141592653589793
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Mon, 19 Sep 2022 17:21:38 +0000'
```

1.11 Help

In REPL, we have access to Python help system. We can get help for modules, classes, functions, etc.

```
>>> help()

Welcome to Python 3.10's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Let's try `help(math)`:

```
    Help on built-in module math:

    NAME
        math

    DESCRIPTION
        This module provides access to the mathematical functions
        defined by the C standard.

    FUNCTIONS
        acos(x, /)
            Return the arc cosine (measured in radians) of x.

        acosh(x, /)
            Return the inverse hyperbolic cosine of x.

        asin(x, /)
            Return the arc sine (measured in radians) of x.
        ...
```

and **sin** function from this module `help(math.sin)`

```
    Help on built-in function sin in math:

    math.sin = sin(x, /)
        Return the sine of x (measured in radians).
```

(type 'q' to leave the help system and return to the REPL.)

To explore properties of modules, classes, functions, etc., we can also use the built-in function **dir**[1]. Called without arguments, it returns the list of names in the current local scope. With an object as the argument, returns a list of attributes for that object. If the object is a module object, the list contains the names of the module's attributes. If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its base classes. Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class' base classes.
For example:

[1] https://docs.python.org/3/library/functions.html?highlight=dir#dir

```
>>> dir()               # attributes in local scope
['__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__']
>>>
>>> import math
>>> dir()               # now 'math' added to the list
['__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'math']
>>> __name__            # name of this module (by default)
>>>
'__main__'
>>>
>>> dir(math)           # attributes of module object 'math'
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
>>> type(math.gamma)    # what's gamma?
<class 'builtin_function_or_method'>
>>> dir(math.gamma)     # attributes of gamma function
['__call__', '__class__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__self__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__text_signature__']
>>> math.gamma.__name__
'gamma'
>>> math.gamma.__doc__  # docstring for gamma function
'Gamma function at x.'
>>> type(math.pi)       # what's pi?
<class 'float'>
>>> math.pi             # value of pi
3.141592653589793
```

Names of many of these attributes begin and end with double underscore. These are “special” names, often of the so called “magic methods”, which we will cover later (see sec. 11.8, p. 234).

1.12 Keywords (reserved words)

As in other languages, some words (names) are reserved for the language itself (like for, while) and cannot be used as names of variables, functions, classes, etc.
These are

Table 1: Python reserved words

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

Chapter 2

Flow control statements

2.1 *pass statement*

The pass statement plays the rôle of an empty statement (like semicolon in C/C++/-Java). The statement does nothing at all, but is useful when syntax requires a statement, but we don't want or need anything.
For example, we can define class which describe some types of exceptions. What we need is just a type of this exception; we don't need to define any methods or fields. We can then leave the definition of such classes empty, as in the example below

Listing 4 AAM-Pass/Pass.py

```python
#!/usr/bin/env python

from math import log10

class ZeroArg(Exception):
    pass

class NegArg(Exception):
    pass

class ZeroDenom(Exception):
    pass

def f(num, denom):
    if denom == 0: raise ZeroDenom
    elif num == 0: raise ZeroArg
    elif num < 0:  raise NegArg
    return log10(num)/denom

for n, d in ( (-10, 2), (1000, 2), (100, 0), (0, 2) ):
    try:
        res = f(n, d)
        print("({:4d}, {}) -> {}".format(n, d, res))
    except Exception as e:
        print("({:4d}, {}) -> exception {}"
              .format(n, d, e.__class__.__name__))
```

which prints

```
( -10, 2) -> exception NegArg
(1000, 2) -> 1.5
( 100, 0) -> exception ZeroDenom
(   0, 2) -> exception ZeroArg
```

2.2 *if statement*

The if statements is similar to analogous constructs known from other languages. The conditions (logical expressions) after if, elif and else do *not* require to be enclosed in parentheses. Remember also to add a colon after each of them:

Listing 5 AAG-Iffs/Iffs.py

```python
#!/usr/bin/env python3

number = int(input("Enter an int "))
if number < 0:
    print("negative numbers do not count")
elif number == 0:
    print("Nothing")
elif number == 1:
    print("One")
elif number == 2:
    print("Two")
else:
    print("Many")
```

Of course, the elif and else clauses are optional, and an else clause, if it has been used, must appear as the last one.

2.3 if-else (conditional) expression

The if-else conditional is not a flow-control statement, but is somehow related to if statement, so we will mention it here (for a more detailed description, see sec. 4.14, p. 60).

It corresponds to conditional expression denoted in C/C++/Java with ternary operator ?:, but the syntax is somewhat different. It's an *expression*, what means it yields a value which can be assigned to a variable or used as a part of another expression. The syntax is

```
expr_if_true if condition else expr_if_false
```

what yields the value of expr_if_true if the condition evaluates to True and expr_if_false otherwise. As is the case with conditional expressions (?:) in other languages, it is short-circuited, i.e., depending of the logical value of condition only expr_if_true or only expr_if_false is evaluated

```
>>> a, b = 5, 0
>>> c = a//b if a > 6 else 3*a
>>> c
15
>>> d = a//b if a < 6 else 3*a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

(in the first case, the condition was false and the expression a//b was not evaluated at all.)

Conditional expressions can be nested, although this should be avoided, as it leads to rather vague expressions

```
>>> for t in (20, 50, 90):
...     print("green" if t < 40 else "yellow" if t < 80 else "red")
...
green
yellow
red
```

2.4 *while loop*

The **while** looks as in other languages. The only difference is that it can optionally contain also an **else** clause which will be executed only when the loop has been exhausted without **break**: if the flow of control exits a loop because of **break**, the **else** clause will *not* be executed. As in other languages, **break** inside a loop means 'go out of this loop' and **continue** means 'go to the next iteration, without completing this one'.

These constructs are illustrated by the program below. Note that the last line will never be executed, because you have to use **break** to exit the main loop:

Listing 6 AAJ-While/While.py

```python
#!/usr/bin/env python

while True:
    number = int(input("Enter an int "))
    if   number == 0:
        break
    if number <= 1:
        print("Your number must be >= 2.")
        continue
    if number in (2, 3, 5, 7):
        print(str(number) + ' is prime.')
    elif number % 2 == 0:
        print(str(number) + ' is not prime.')
    else:
        i = 3
        while i*i <= number:
            if number % i == 0:
                print(str(number) + ' is not prime.')
                break
            i += 2
        else:
            print(str(number) + ' is prime.')
else:
    print('This will never be printed')
```

2.5 *for loop*

The form of the for loop resembles the so called 'for-each' (or 'ranged for') loop from other languages. The syntax is

```
for target_list in expression_list:
    body_of_the_loop
[else:
    block_of_code]
```

where the else clause is optional and is executed only when the loop has been exhausted without encountering any break or return statement (as is also the case for the while loop). The expression_list is evaluated first (and once only) — it should yield an **iterable** object.

Iterables are objects which provide the __iter__ function which in turn returns an **iterator**.[1] Instead of calling __iter__ on an object, one can use build-in function iter with an iterable object as the argument. In some situations, if the object provides the __getitem__ method, this function can "produce" an iterator without calling __iter__.

Iterator in turn is an object which provides the __next__ function returning the next element of a sequence (again, one can use the built-in function next with an iterator as the argument). When the sequence is exhausted, __next__ should raise (throw) a StopIteration exception. An iterator should also possess the __iter__ method which just returns itself (this makes iterators themselves iterables).

The for loop takes care about all this automatically. Having the iterable (provided by expression_list), iter is invoked to get the iterator. The body of the loop is then executed for each item provided by the next function. Each item is assigned to the target_list. The target_list doesn't have to be a single value if items returned by the iterator are, e.g., key-value pairs of a dictionary or tuples, as in the example below. At some point next raises the StopIteration exception, which is handled by the for loop, so we don't see it.

In the example below, we use a for loop and then we repeat the same action "by hand", calling iter, and then next until we get an exception:

Listing 7 AAK-For/For.py

```python
#!/usr/bin/env python

persons = [("Ann", 23), ("Bea", 34), ("Sue", 29)]
for name, age in persons:
    print('{} is {} years old'.format(name, age))

print('\n' + '*'*5 + " now 'by hand' " + '*'*5 + '\n')

it = persons.__iter__()
            # or it = iter(persons)

a = it.__next__()
print('{} is {} years old'.format(a[0], a[1]))

```

[1] Like Java objects implementing the Iterable interface, which declares abstract method iterator returning an object of a class implementing Iterator.

```python
a = next(it) # equivalently like this
print('{} is {} years old'.format(a[0], a[1]))

a = next(it)
print('{} is {} years old'.format(a[0], a[1]))

a = it.__next__()
print('{} is {} years old'.format(a[0], a[1]))
```

The program prints

```
Ann is 23 years old
Bea is 34 years old
Sue is 29 years old

***** now 'by hand' *****

Ann is 23 years old
Bea is 34 years old
Sue is 29 years old
Traceback (most recent call last):
File "/home/werner/python/py_progs/sources/AAK-For/For.py",
        line 21, in <module>
    a = it.__next__()
StopIteration
```

And another example, this time with a dictionary

Listing 8 AAN-DictLoop/DictLoop.py

```python
#!/usr/bin/env python

fourSeasons = {'Spring' : 'E major', 'Summer' : 'G minor',
               'Autumn' : 'F major', 'Winter' : 'F minor'}

for key, value in fourSeasons.items():
    print(key + ' is in the key of ' + value)
```

which prints

```
Spring is in the key of E major
Summer is in the key of G minor
Autumn is in the key of F major
Winter is in the key of F minor
```

Note that there is no variable denoting the index of the element provided by the iterator (as in traditional **for** loops in C/C++/Java). Very often such index would be useful, though, so there is a built-in function **enumerate** which takes an iterable and returns an *enumerate* object which can be iterated over and provides 2-tuples, the first element of which is the index and the second is an item that would be returned by a “normal” iterator; for example

Listing 9 AAL-ForEnum/Enum.py

```
#!/usr/bin/env python

lst = [1, 7, 4, 9, 2, 6, 3]
indsOdd = []
for i, e in enumerate(lst):
    if e % 2 != 0:
        indsOdd += [i]
print('Odd elements at positions',indsOdd)
```

prints the indices of all odd elements of the list lst

```
Odd elements at positions [0, 1, 3, 6]
```

Indices returned by an enumerate object start from zero, but this can easily be changed by providing additional argument to **enumerate**:

Listing 10 AAO-EnumStart/EnumStart.py

```
#!/usr/bin/env python

seasons = [ "Spring", "Summer", "Autumn", "Winter" ]
for ind, s in enumerate(seasons, 1):
    print('Season', ind, '->', s)
```

which prints

```
Season 1 -> Spring
Season 2 -> Summer
Season 3 -> Autumn
Season 4 -> Winter
```

Indices can also be obtained by using a built-in function **range**, which returns a *range object* — an iterable representing an arithmetic progression of integer numbers. Such sequence may be created by the `range(start, stop, step)` expression: the first number of the sequence will be start (default is 0), the difference between consecutive numbers will be step (default is 1) and the last element will be the last which is still smaller than stop (still greater than stop for negative steps). Special version of this function, `range(n)`, with only one argument, is equivalent to **range(0, n, 1)**. Here we show the “anatomy” of using **range** and then just the **for** loop which takes care about everything...

```
>>> r = range(1, 6, 2)
>>> type(r)
<class 'range'>
>>> len(r)
3
```

```
>>> hasattr(r,'__iter__')
True
>>> it = iter(r)
>>> hasattr(it,'__next__')
True
>>> it.__next__()
1
>>> next(it)
3
>>> next(it)
5
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>> for i in r:     # everything taken care of by 'for'
...     print(i)
1
3
5
```

(built-in function **hasattr** checks if an object has a given attribute).

2.6 match statement

match statement corresponds to **switch** in Java or C++, but is richer in features, resembling similar constructs in Haskell, Rust or Scala.

In the simplest form it's easy to interpret and understand if we remember **switch** from C/C++/Java:

Listing 11 AAQ-MatchBasic/MatchBasic.py

```python
#!/usr/bin/env python

k = 3
match k:
    case 0:
        print('zero')
    case 1 | 2 | 3 | 4:          ❶
        print('less than 5')
    case 5:
        print('five')
    case _:                      ❷
        raise ValueError()
```

Note that conditions in case-clauses can be "ORed" (❶) using the vertical bar |. Instead of a **default** clause, as in other languages, one can use "catch all" clause ❷, traditionally denoted by _ (underscore) symbol.

Note also that there is no “fall through” here: only the first matching **case** clause is executed (or none, if there is no matching **case** clause).

We can match not only values, but also structured data; moreover, it is possible to assign values found in matched structures to variables that can be then used in a corresponding **case** clause.
In the example below, consecutive **case** clauses correspond to data of different lengths (these could be tuples or lists). Note the notation used in line 11: the `*d` symbols allows us to use `d` as a sequence containing all the remaining (except the first which is bound to name `c`) elements of the matched sequence.

Listing 12 AAT-MatchLen/MatchLen.py

```
 1  #!/usr/bin/env python
 2
 3  tuples = [ (1, 2, 3), ['a', 'b'], (4, 5, 6, 7, 8), "One" ]
 4
 5  for item in tuples:
 6      match item:
 7          case [a1, a2]:
 8              print('Two elements', a1, a2)
 9          case [b1, b2, b3]:
10              print('Three elements', b1, b2, b3)
11          case [c, *d]:
12              print('Many elements', c, end=' ')
13              for x in d:
14                  print(x, end=' ')
15              print()
16          case e:
17              print('One element', e)
```

The program prints

```
Three elements 1 2 3
Two elements a b
Many elements 4 5 6 7 8
One element One
```

The list notation is used here for sequences of different lengths, but instead of, for example, `case [a1, a2]:`, we could have used `case (a1, a2):` or even just `case a1, a2:` with no brackets at all.

We can also match structure and values at the same time. In the example below, the first element of `cmd` is matched to a literal string, while the remaining are bound to variables then used in the corresponding **case** clause

Listing 13 AAV-MatchVS/MatchValStr.py

```
#!/usr/bin/env python

commands = (
             ('store', 'MyTable', 'John', 'Doe', 123),
             ('send',  'Hello John', 'john@somewhere'),
             ('print', 'Hello, World!'),
             'Error no 42'
           )

def printMessage(message):
    print('Printing message:', message)

def sendMessage(message, address):
    print("Sending '" + message + "' to", address)

def storeInDB(database, *args):
    print('Storing in DB:', end=' ')
    for a in args:
        print(a, end=' ')
    print()

def printError(error):
    print("ERROR: couldn't process '" + error + "'")


for cmd in commands:
    match cmd:
        case "print", message:
            printMessage(message)
        case ("send", message, address):
            sendMessage(message, address)
        case ["store", database, *data]:
            storeInDB(database, *data)
        case someError:
            printError(someError)
```

The program prints

```
Storing in DB: John Doe 123
Sending 'Hello John' to john@somewhere
Printing message: Hello, World!
ERROR: couldn't process 'Error no 42'
```

In the last example one `case` clause “catches” all two-element sequences bounding both elements to variables and then processes them (adding to appropriate collections); anything that is not a two-element sequence is handled differently:

Listing 14 AAX-MatchSN/MatchSeqNam.py

```
#!/usr/bin/env python

fruit   = {}       # empty dictionary
veggies = dict()   # also empty dictionary
```

```
others  = dict()
noprice = set()

frunames = ["Apple", "Pear"]
vegnames = ["Carrot","Beetroot" ]

shopping = [
             ("Carrot", 4), ("Pear", 7),   "Hammer",
             ("Tulip", 6),  ("Carrot", 5), ("Pear", 3),
             "Chocolate",   ("Apple", 8),  ("Carrot", 4),
             ("Apple", 2),  "Laptop",      ("Beetroot", 1)
           ]

for item in shopping:
    match item:
        case [fruveg , price]:
            aux = (fruit if fruveg in frunames
                         else veggies if fruveg in vegnames
                                      else others)
            if fruveg not in aux: aux[fruveg] = 0
            aux[fruveg] += price
        case justName:
            noprice.add(justName)

print('Fruit:   ', fruit)
print('Veggies: ', veggies)
print('Others:  ', others)
print('No price:', noprice)
```

The program prints

```
Fruit:    {'Pear': 10, 'Apple': 10}
Veggies:  {'Carrot': 13, 'Beetroot': 1}
Others:   {'Tulip': 6}
No price: {'Laptop', 'Hammer', 'Chocolate'}
```

More details can be found in PEP 636.[1]

[1] http://peps.python.org/pep-0636

Chapter 3

Basic data types

Basic data types which are built-in into the language itself are

- boolean
- numbers (integers, floating-point and complex). There are also useful numeric types in the library: fractions.Fraction and decimal.Decimal;
- sequences (lists, tuples, strings, types bytes, bytearray, memoryview);
- sets;
- dictionaries (maps).

Note that there are no classic arrays here but there are lists, sets and maps (called dictionaries in Python) which in other languages belong to the standard libraries rather than to the language itself.

As everything in Python is an object, there are other built-in types describing

- modules,
- classes and class instances (objects),
- functions,
- methods,
- code objects,
- type objects,
- the None object,
- the Ellipsis object,
- the NotImplemented object,
- internal objects.

Many builtin types (int, float, str) have converting constructors, e.g., int('123') yields integer 123, while str(123) gives the string '123'.

3.1 NoneType type

There is a special type denoting ‘nothing’ — only one singleton object of this type is created; its name is None. It has no attributes and evaluates to False in Boolean context. None is returned by functions that don’t explicitly return any value (i.e., correspond to void functions in C/C++/Java).

```
>>> type(None)
<class 'NoneType'>
>>> a = print()

>>> b = print()

>>> type(a), type(b)
(<class 'NoneType'>, <class 'NoneType'>)
>>> id(a), id(b)
(94793807807456, 94793807807456)
>>> if a:
...     print('true')
... else:
...     print('false')
...
false
```

None is also often used as the default value of optional arguments, so the function can easily detect whether the caller has actually passed a value for that argument or not.

3.2 ellipsis type

A special object of type **ellipsis**, which may be written as **Ellipsis** or **...** (three dots), is used almost uniquely in numerical packages for slicing in multidimensional arrays (matrices). In Boolean context, it evaluates to **True**.

```
>>> type(Ellipsis)
<class 'ellipsis'>
>>> type(...)
<class 'ellipsis'>
>>> id(Ellipsis), id(...)
(9422880, 9422880)
>>> if Ellipsis: print('T')
... else:        print('F')
...
T
```

3.3 Typ NotImplementedType

A special type with one singleton object named **NotImplemented**. It is used as the return value in special methods which should exist but haven't been implemented yet. It is not recommended (and considered obsolete) to use this object in a logical context.

3.4 Numeric types

Numeric types describe numbers of integer and floating-point types. We will include here also the Boolean type **bool**, because it inherits from **int** and *can* be used (unfortunately) in numerical contexts.

Generally, Python is rather reluctant to implicitly convert from one type to another. However, as in most other languages, you can mix numeric values of different types when operating on them. When you add an integer to a float, for example, the result will be float (as "wider" than **int**)

```
>>> a = 2; b = 2.5
>>> a+b, type(a+b)
(4.5, <class 'float'>)
```

Similarly, **complex** type is wider than **float**, so in operations involving complex numbers, integers and floats are promoted to **complex**:

```
>>> a = 1; b = 1.5; c = (1 + 1j)
>>> d = a + b + c;
>>> d, type(d)
((3.5+1j), <class 'complex'>)
```

In order to facilitate mixed-type (**int**, **float**, **complex**) arithmetic, floats and integers do have their real and imaginary attributes real and imag defined (their values are, of course, equal to 0):

```
>>> a = 7
>>> a, type(a), a.real, a.imag
(7, <class 'int'>, 7, 0)
>>> b = 3.75
>>> b, type(b), b.real, b.imag
(3.75, <class 'float'>, 3.75, 0.0)
```

Constructors of numeric types may be used to explicit conversions (from other numeric type or strings).
Casting **float**→**int** rounds towards zero:

```
>>> a = 3.9; b = -3.9
>>> int(a), int(b)
(3, -3)
```

and you can also use strings:

```
>>> int('123'), float('-1.2e+2'), complex('2-3j')
(123, -120.0, (2-3j))
```

Note that for complex numbers, the string must not contain spaces:

```
>>> complex('1 + 1j')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

3.4.1 Integers

There is only one integer type – **int**. However, the size of the representable integers is not limited: Python will transparently select an appropriate internal representations for any, arbitrary large value; e.g., 33^{33} will be calculated correctly, giving

$$129110040087761027839616029934664535539337183380513$$

(operating on such numbers may be very inefficient, though).
Integer literals can be prefixed with a digraph denoting the base: 0b or 0B for binary, 0o or 0O for octal, and 0x or 0X for hexadecimal. For example,

$$47 \equiv 0b101111 \equiv 0o56 \equiv 0x2F.$$

As in Java, underscores can be used in integer literals to separate groups of digits

```
>>> 1_234_567
1234567
```

The **int** class has constructors which can be used as conversion function: in particular, there are such conversions from strings (but also from **bytes** or **bytearray**). The second, optional, argument specifies the radix (of course, the default is 10):

```
>>> int('1D', 16)
29
>>> int('1D', 16), int('0x1D', 16)
(29, 29)
```

```
>>> int('57', 8), int('0o57', 8)
(47, 47)
>>> int('11001',2), int('0b11001', 2)
(25, 25)
>>> int('123', 10), int('123')
(123, 123)
>>> int('2Z', 36)
107                    # 2*36 + 35
```

There are also built-in functions **bin**, **oct** and **hex** for **int** → **str** conversions, where the resulting string represents the given number with radix 2, 8 or 16 (for decimal representation, just use the constructor of **str**):

```
>>> a = bin(0x8E)
>>> a, type(a)
('0b10001110', <class 'str'>)
>>>
>>> b = oct(0xF)
>>> b, type(b)
('0o17', <class 'str'>)
>>>
>>> c = hex(255)
>>> c, type(c)
('0xff', <class 'str'>)
>>>
>>> d = str(123)
>>> d, type(d)
('123', <class 'str'>)
```

Implementation of the **int** type is rather unique in Python. All integers, small and arbitrary big, are represented by objects of the same type – **int**; no **long**s, **short**s, signed and unsigned types, etc.[1]

Objects representing small integers, at least in the range $[-5, 256]$, are created once only and cached, so all references to their values point to exactly the same objects:

```
>>> a = -2; b = 0b1011 - 0xD; c = -4000//2000
>>> a, b, c
(-2, -2, -2)
>>> id(a), id(b), id(c)
(9801152, 9801152, 9801152)
```

Integers are internally represented by C-structures **PyLongObject** containing, among other fields,[2]

- pointer to array ob_digit of 'digits', which are just **uint32_t** numbers;
- size of this array ob_size.

For numbers smaller than 2^{30}, the array has only one element which is interpreted directly as an unsigned four byte integer. For bigger numbers, elements of the array ('digits') are interpreted as coefficients at increasing powers of 2^{30} and themselves have

[1] But see sec. ??, p. ??.

[2] On 64-bit platforms.

values in the range $[0, 2^{30} - 1]$. For example,

$$398540651359084385821447815 = 1234567 + 23456789 * 2^{30} + 345678912 * 2^{30*2}$$

so for this number, ob_digit is $[1234567, 23456789, 345678912]$ and ob_size is 3.

Negative numbers are represented in the same way and their reversed sign is signaled only by negative value of ob_size. If ob_size is 0, the number is 0.

Due to their complicated implementation, objects of type **int** occupy different amount of memory, depending on their value. Minimum size of these objects is 28 bytes (3.5 times the size of a C/C++/Java **long**). This happens when the array ob_digit of size 1 is sufficient (i.e., for numbers smaller than 2^{30}). However, for bigger numbers, the size of this array must grow and objects become larger:

```
>>> a = 1
>>> a, type(a), a.__sizeof__()
(1, <class 'int'>, 28)
>>>
>>> a = 2**30-1
>>> a, type(a), a.__sizeof__()
(1073741823, <class 'int'>, 28)  # still one element
>>>
>>> a = a + 1
>>> a, type(a), a.__sizeof__()
(1073741824, <class 'int'>, 32)  # two elements needed
>>>
>>> a = (2**30-1)*2**30 + (2**30 - 1)
>>> a, type(a), a.__sizeof__()
(1152921504606846975, <class 'int'>, 32) # still two elements
>>>
>>> a = a + 1
>>> a, type(a), a.__sizeof__()
(1152921504606846976, <class 'int'>, 36) # now three needed
```

There are, of course, algorithms for arithmetic operations on numbers represented in this way.

3.4.2 Boolean type

Python supports special type for Boolean values – **bool** (which is a subclass of **int**). It has only two values: True and False. However, as in many other languages (but not Java!), values of other types may be used in contexts requiring a Boolean value; the general rule is that the following objects

- numerical values equal to zero,
- empty sequences and collections (lists, sets, dictionaries, strings, tuples),
- special object None,
- constant False

are all "falsy" — they are interpreted as False in Boolean contexts. All other objects are "truthy". The only object that shouldn't be used in the logical context is **NotImplemented** (which is, however, evaluated to True).

If we really want something "truly" Boolean, we can always use conversion to **bool** just by passing an object to the **bool**'s converting constructor:

```
>>> a = bool("Hello")
>>> a, type(a)
(True, <class 'bool'>)
>>> bool({})
False
>>> bool(0)
False
>>> bool(78)
True
>>> bool(NotImplemented)
<stdin>:1: DeprecationWarning: NotImplemented should
           not be used in a boolean context
True
>>> bool(Ellipsis)
True
>>> bool([[]]) # non-empty list: contains one element!
True
```

Unfortunately, objects of type **bool** are represented by integers (0 and 1), so formally they *are* of a numeric type (although they shouldn't be...). For example,

```
>>> True/3
0.3333333333333333
>>> (7-False)/(True+1)
3.5
```

This fact that **bool** inherits from **int** has sometimes unexpected consequences; for example

```
>>> True == 2-1
True
>>> True == 3-1
False
```

and it's impossible to have both **True** and 1 in a set or as keys in a dictionary.

Logical alternative and conjunction are denoted by the **or** and **and** keywords, respectively. As in most other languages, they are both short-circuit operators – the second operand is evaluated if, and only if, the result cannot be established after evaluation of the first one, i.e., only when the first operand is interpreted as **False** for alternative or as **True** for conjunction.

Logical negation is denoted by the **not** keyword.

Note that `||`, `&&` and `!`, known from other languages, will *not* work in Python.

3.4.3 Floating point numbers

There is only one floating point type – **float**. It corresponds to what is called **double** in, for example, C/C++/Java. These numbers occupy 8 bytes (64-bit) with 1 sign bit, 11 bits for exponent part and 52 bits for the mantissa; this gives 53 binary-digits precision, because in binary system the most significant digit of any number must be 1, so there is no need to store it physically in memory (for details, see, e.g., description here[1]).

[1] https://en.wikipedia.org/wiki/IEEE_floating_point

Remember that floating point numbers are almost always *approximations* only; as numbers are internally expressed in binary system, even such “innocent” values like 0.1 do not have exact representation. Only rational numbers with a power of 2 in the denominator may be represented exactly:

```
>>> v0100 = 0
>>> v0125 = 0
>>> for i in range(1_000_000):
...     v0100 += 0.100
...     v0125 += 0.125
>>> v0100 - 100_000
1.3328826753422618e-06
>>> v0125 - 125000
0.0
```

In the second case, we were adding 0.125, which is $1/2^3$ and so *is* represented exactly and sums up to the correct value 125 000. However, there is no such number like 0.1, so adding it milion times does not give us the correct sum 100 000: we can see this by printing 0.1 as a **Decimal**:

```
>>> import decimal
>>> decimal.Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

The above result is *exact* — it is, in decimal system, exactly the number represented by *approximation* of 0.1 on 53 binary digits of a **double**; this value in binary system has no finite representation, much like 1/3 in decimal system.

Sometimes, in particular in numerical computations, one has to deal with infinities (Inf) or not-a-number (NaN) values; these are also defined in Python:

```
>>> a = float('nan')
>>> b = float('nan')
>>> c = a + b
>>> a == b, a, b, c
(False, nan, nan, nan)
```

As we can see, NaN compared with anything, even with itself, gives **False**, and any arithmetic operation involving a NaN yields also a NaN.
Let us look at infinities:

```
>>> x = float('inf')
>>> y = float('-inf')
>>> x, y, x + 100, x - y, x + y
(inf, -inf, inf, inf, nan)
```

Adding/subtracting anything finite to/from ±Inf gives ±Inf; also infinities of the same sign add up to infinity. However, subtracting infinities (i.e., adding infinities of opposite signs) is not mathematically defined, so it yields a NaN.

3.4.4 Complex numbers

Literals of the **complex** type are written with the letter ’j’ (or ’J’, but *not* ’i’, as in mathematics) appended to the imaginary part.
For example `0.5*(1+1j)**2` gives `1j`. Both real and imaginary parts are represented by 64-bit **float**s and can be accessed as read-only attributes real and imag:

```
>>> z = 3+4j
>>> z.real
3.0
>>> z.imag
4.0
>>> z.imag = 8
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  AttributeError: readonly attribute
```

The **abs** function (which is built-in and doesn't require any imports) works also for complex values; moreover, the ***cmath*** module contains many other mathematical functions that operate on and return complex numbers (like **phase**, **exp**, **log**, **sqrt**, trigonometric and hyperbolic functions, etc.). One can mix **float**s and **int**s with **complex** numbers in all mathematical operations: if there is at least one complex operand, the others will be promoted to **complex** and the result will also be of that type (perhaps with imaginary part equal to 0):

```
>>> z = 1 + 1j
>>> import cmath
>>> 2 + cmath.log(z)/1.5
(2.2310490601866486+0.5235987755982988j)
>>> from cmath import exp as cexp
>>> z - cexp(cmath.log(z))
2.220446049250313e-16j       # essentially 0
```

3.5 Strings (introduction)

Literal strings (strings are of type **str** in Python) may be written in several ways. They can be enclosed by single apostrophes or double quotes. If the apostrophes are used, quotation mark inside behaves as a normal character and doesn't need to be escaped, while the apostrophe itself has to be escaped (and the other way around for strings enclosed by double quotes). Two adjacent literal strings are concatenated and represent one string, as in C/C++

```
>>> "That's OK, " 'double quote doesn\'t have to be "escaped" here'
'That\'s OK, double quote doesn\'t have to be "escaped" here'
```

One can insert into a string any character knowing its Unicode code. To enter a character with 16-bit Unicode code point, notation `\uxxxx` is used; for 32 bit values, substitute capital 'U' for 'u' (`\Uxxxxxxxx`). Each 'x' here stands for one hexadecimal digit.

```
>>> '\u00a9 \u017b\u00f3\u0142w'
'© Żółw'
```

As in most other languages, certain special characters can be entered by escaping one character with the backslash:

- `\a` for bell character (BEL);
- `\b` for backspace (BS);
- `\f` for formfeed (FF);
- `\n` for new line a.k.a. line feed (LF);

- `\r` for carriage return (CR);
- `\t` for horizontal tab (HT);
- `\v` for vertical tab (VT);
- `\'` for single quote (apostrophe);
- `\"` for double quote.

It is also possible to delimit a string with triple quotes (single or double). Inside such strings, escape characters work as usual, but apostrophes and double quotes do not need to be escaped; also, new-line characters are preserved

```
>>> """That's OK, quotes don't
... have to be "escaped" here"""
'That\'s OK, quotes don\'t\nhave to be "escaped" here'
```

By prefixing a string with the letter 'r' (or 'R'), we get something known as a **raw string**. In such strings, the backslash is taken literally, not as the escape character, so string `r"\t\v\b"` contains 6 characters, three of them backslashes and three letters (without the 'r', it would be a three-character string containing a tabulator, a vertical tabulator and a backspace).
Raw strings are especially helpful when defining regular expressions, which usually contain lots of backslashes.

Important: As in Java, strings (objects of type **str**) are *immutable*. It is not possible to modify a string: functions which seemingly do it are in fact returning a new string based on an existing one:

```
>>> s = "Alice"
>>> id(s)
139733635023176
>>> s = s.lower()
>>> s, id(s)
('alice', 139733635017168)
```

Chapter 4

Operators

As we know from other languages, like C++/Java, operators are really functions in disguise, referred to not by “normal” name but rather symbols (*, &, +, etc). placed in front of or between operands (arguments). We know also that there are unary (!, ~, etc.), binary (+, %, etc.) and even one ternary operator ?:.

In this chapter, we will present an overview of operators in Python, their features and precedence. But first a few words about assignment, which is *not* an operator in Python.

4.1 Assignment (not an)operator

Operators, unary or binary, when applied to their operands, always, by definition, yield values. Note, however, that in Python, assignment does *not* return any value, it’s not an expression

```
>>> a = (b = 2)              # OK in C/C++/Java
File "<stdin>", line 1
    a = (b = 2)
        ^
SyntaxError: invalid syntax
```

It follows that assignment =, but also augmented forms like +=, *=, etc., are *not* operators, as they are in C/C++/Java, where assignment is a “normal”, binary, right-associative operator.

What can be surprising, though, `a = b = 2` *is* valid! Chained assignments are treated in a special way: first the rightmost expression (it has to be an expression!) is evaluated and then its value is, one by one, *from left to right* assigned to “targets” in front of it:

```
>>> b = [3, 3]
>>> a = b[a] = b[a-1] = len(b)-1
>>> b
[1, 1]
```

As we can see, when the value of `len(b)-1`, which is 1, is assigned to `b[a]`, variable `a` already exists and has value 1. Thus, conceptually,

$$\text{target}_1 = \text{target}_2 = \text{target}_3 = \text{expr}$$

is equivalent to evaluating expr and then assigning

$$\begin{aligned}\text{target}_1 &= \text{expr}\\ \text{target}_2 &= \text{expr}\\ \text{target}_3 &= \text{expr}\end{aligned}$$

in this order.

However,

```
>>> x = y = z+= 2
SyntaxError: invalid syntax
```

doesn’t work, because the rightmost part, `z += 2`, is *not* an expression — it has no value.

4.2 Assignment (true) operator

In Python 3.8 the "true" assignment (assignment expression) was added to the language. That means that such assignment does have the type and value of the left-hand side after the assignment, as it does in C/C++/Java.
It is denoted by := symbol[1] and has the form `name := expr`, where `name` is an identifier and `expr` is any expression (except an unparenthesized tuple).

Such assignments often simplify the code and make it more readable.

For example, suppose we read data from the user until they enter `"quit"` (and we add the data read to a list). We would have to write something like this:

```
numbers = []
inp = input("Enter number (or 'quit'): ")
while inp != "quit":
    numbers.append(int(inp))
    inp = input("Enter number (or 'quit'): ")
```

This works, but is ugly — the line with **input** is repeated twice. Now we can do better using the *walrus* operator :=

```
numbers = []
while (inp := input("Enter number (or 'quit'): ")) != "quit":
    numbers.append(int(inp))
```

Assignment expressions are commonly used to give a name to an intermediate result, so it can be used without the need to recalculate it, even in the same statement:

```
s = 'rotator'
if (r := s[::-1]) == s:
    print('Palindrom')
else:
    print('Not palindrom -> ' + r)
print('r is ' + r)
```

This program prints

```
Palindrom
r is rotator
```

what shows another important feature of Python: although `r` is created inside if statement, it is still visible after the if (see discussion of scopes, sec. 5.1, p. 63).

Assignment expressions can also be used in comprehensions (sec. 6.1, p. 93): for example

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

print([(n, r) for n in range(24,31) if (r := fib(n)) % 2 != 0])
```

[1] Called 'walrus'

prints odd Fibonacci numbers F_n for $n \in [24, 30]$

```
[(25, 75025), (26, 121393), (28, 317811), (29, 514229)]
```

We will encounter other uses of the walrus operator in the course of the lecture.

4.3 Operators and operator precedence

The precedence of Python operators is presented in the table below:

Table 2: Operator precedence

Operator(s)	Description
(expressions...), [expressions...], {key: value...}, {expressions...}	Parenthesized expressions: literal lists, sets, dictionaries
x[i], x[i:j], x[i:j:ks], x(arguments...), x.attribute	Subscription, slicing, call operator, attribute reference
await x	Await expression
**	Exponentiation
+x, −x, ∼x	Unary +/-, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, modulo
+, −	Addition, subtraction
<<, >>	Bitwise shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not	Boolean NOT
and	Boolean AND
or	Boolean OR
if-else	Conditional expression
lambda	Lambda expression
:=	Assignment expression

4.4 Parenthesized expressions

Parenthesized expressions are used to create lists (`[ expressions... ]`) (sec. ??, p. ??), sets (`{ expressions... }`) (sec. 7.3, p. 130), dictionaries (`{ key:value... }`) (sec. 7.4, p. 137):

```
>>> a = [1, 2, 'Hello']
>>> a, type(a)
([1, 2, 'Hello'], <class 'list'>)
>>> b = {1, 2, 'Hello'}
>>> b, type(b)
({1, 2, 'Hello'}, <class 'set'>)
>>> c = {1 : 'un', 3 : 'trois', 'un' : 1, 'trois' : 3}
>>> c, type(c)
({1: 'un', 3: 'trois', 'un': 1, 'trois': 3}, <class 'dict'>)
```

Instead of specifying a sequences of elements, one can use comprehensions for lists (sec. 6.1.1, p. 94), sets (sec. 6.1.2, p. 97), dictionaries (sec. 6.1.3, p. 98) and generators (sec. 6.2.1, p. 102):

```
>>> a = [x*x for x in range(1, 11) if x%2 != 0]
>>> a, type(a)
([1, 9, 25, 49, 81], <class 'list'>)
>>>
>>> b = {x for x in dir(a) if x[:1] == 'c'}
>>> b, type(b)
({'count', 'copy', 'clear'}, <class 'set'>)
>>>
>>> c = {x : len(x) for x in b}
>>> c, type(c)
({'count': 5, 'copy': 4, 'clear': 5}, <class 'dict'>)
>>>
>>> d = (x for x in range(1,101) if x**2 % 10 == 6)
>>> d, type(d)
(<generator object <genexpr> at 0x7f52a900>, <class 'generator'>)
>>> for x in d:
    print(x, end=' ')

4 6 14 16 24 26 34 36 44 46 54 56 64 66 74 76 84 86 94 96
```

4.5 Subscription and slicing

4.5.1 Indexing

Indexing in Python behaves as in C/C++/Java — index is given in square brackets and indexing starts from 0. Indices can be used for objects of types implementing the **__getitem__** method (although it's not enough for object to be a full-fledged sequence). In particular, all sequences do support indexing (**list**s, **tuple**s, **range**s, strings).

```
>>> ls = [1, 2, '3', 'four', ('Hello', 'World')]
>>> ls[0]
1
>>> ls[3]
'four'
>>> ls[4]
('Hello', 'World')
>>> ls[4][1]
'World'
```

This is not surprising, but Python has more to offer. Sometimes, it's more convenient for us to specify an index counting from the end; for example "I want the third last element of this sequence". Instead of calculating which index it corresponds to, we can use *negative* indices.

Index -1 corresponds to the *last* element, and so is equivalent to positive index $-1 + \text{leng}$, where leng is the number of elements in our sequence. Index -2 corresponds to the second last element, and so on. Generally, negative index $-n$ corresponds to positive index $-n + \text{leng}$. For example, the string `'PYTHON'` has length 6 and its elements (characters) can be indexed as follows:

```
 0   1   2   3   4   5
 P   Y   T   H   O   N
-6  -5  -4  -3  -2  -1
```

Hence, legal indices are

$-\text{leng}, -\text{leng}+1, \ldots, -1, 0, 1, \ldots, \text{leng}-1$

what can be seen from the following snippet:

```
>>> s = 'PYTHON'
>>> s, len(s)
('PYTHON', 6)
>>> s[0]
'P'
>>> s[5]
'N'
>>> s[-1]
'N'
>>> s[-6]
'P'
>>> s[6]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
>>> s[-7]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

A very common task is to find the index of an element of a sequence (list, tuple, string,...). There is a useful method **index** which allows to do it (it finds the first occurrence of an element in a sequence):

```
>>> a = (1, 4, 5, 8)     # tuple
>>> b = [1, 4, 5, 8]     # list
>>> c = '1458'           # string
>>> a.index(5), b.index(5), c.index('5')
(2, 2, 2)
```

One can add an additional argument which indicates from which index to start the search

```
>>> a = [5, 6, 7, 5, 6, 7]
>>> a.index(5, 2)
>>> 3
```

The result is 3 because searching for 5 starts here from the element with index 2 (and value 7). Optionally, one can add one more argument indicating the last index defining a half-open range of elements which will be searched (as usual, the element with this last index is *excluded* from this range):

```
>>> a = [5, 6, 7, 8, 6, 4, 6]
>>> a.index(6, 2, 5)
4
```

Here, the value 6 was searched in a range of indices $[2, 6)$, i.e., $2, 3, 4, 5$ (without 6). If there is no element with a given value, **ValueError** exception is thrown:

```
>>> a = [1, 2, 4, 5, 3]
>>> a.index(3)
4
>>> a.index(3, 0, 4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 3 is not in list
```

Exception has been raised here, because the value 3 is not in the range of indices $[0, 4)$, which does not include index 4. In order to avoid exceptions, we can first check if the element we are looking for occurs in a given sequence at all (**in** and **not in** operators):

```
>>> a = [1, 2, 3, 4]
>>> b = 'Hello'
>>> 2 in a
True
>>> 5 in a
False
>>> 5 not in a
True
>>> 'el' in b
True
```

Analogous method, **find**, works for strings and objects of type **bytes** (see sec. **??**, p. **??**); instead of raising an exception, it returns -1 if a value searched for has not been found.

```
>>> s = 'abcdefg'
>>> s.find('d')
3
>>> s.find('h')
-1
>>> s.find('b', 2)     # start from index 2
-1
```

4.5.2 Slices

Slices are *copies* of subsequences of sequences (taking a slice does not change the original sequence).

Suppose seq is a sequence and it's length (number of elements) is leng. Then after

```
sl = seq[beg:end:step]
```

sl will be a sequence of *copies of* elements (don't forget these are *references*!) of seq corresponding to indices $\mathrm{beg}, \mathrm{beg}+\mathrm{step}, \mathrm{beg}+2*\mathrm{step}$, *ldots* up to, but **not** including, end.
For example, for [2:7:1] these will be elements with indices $2, 3, 4, 5, 6$, and for [2:7:2] — $2, 4, 6$.
If step is not specified, it defaults to 1, so [2:5] is equivalent to [2:5:1]. One can omit beg (but not a colon that follows it) and/or end; then, for positive step:

- beg defaults to 0, so [:5] is equivalent to [0:5:1],
- end defaults to the length of the sequence.

For example:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f']
>>> ls[:3]                      # = ls[0:3:1]
['a', 'b', 'c']
>>> ls[1::2]                    # = ls[1:6:2]
['b', 'd', 'f']
>>> ls[:]                       # = ls[0:6:1]
['a', 'b', 'c', 'd', 'e', 'f']
>>> ls[2:6:2]
['c', 'e']
>>> ls[5:1:-1]
['f', 'e', 'd', 'c']
```

In the last example, step is negative, but both beg and end are explicitly specified.
But negative steps become tricky when beg and/or end are omitted. Then

- beg defaults to -1,
- end defaults to -leng-1.

For example, taking a string as our sequence (note that its length is 6, so the default value of end is -7):

```
>>> s = 'abcdef'
>>> s[:-2:-1]     # -1
'f'
>>> s[-3:-6:-1]   # -3, -4, -5
'dcb'
>>> s[:-4:-1]     # -1, -2, -3
'fed'
>>> s[-2::-1]     # -2, -3, -4, -5, -6
'edcba'
>>> s[:-5:-2]     # -1, -3
'fd'
>>> s[::-2]       # -1, -3, -5
'fdb'
>>> s[::-1]       # -1, -2, -3, -4, -5, -6
'fedcba'
```

In comments, after the # symbol, indices that are included in the resulting slice are shown. They are negative, but we remember that they are equivalent to positive indices after adding the length of the sequence (6 in our case). Remember also that the index end is never included.

The last example is interesting: as we can see, the slice `[::-1]` yields a copy of the entire sequence, but in reverse order![1]

In order to see better what is going on, we can use objects of type **slice**. As a matter of fact, when we use something like `seq[beg:end:step]`, under the hood object of this type will be created and used to find "real" (non-negative) indices. We can do it "by hand" (arguments **None** of the constructor mean "take the default"):

```
>>> ls = [1, 2, 3, 4, 5]
>>> sl = slice(None, 3, 2)
>>> ls[sl]
[1, 3]
```

As we noted, `seq[::-1]` returns the copy of `seq` in reverse order. Let's see how the indices are calculated. Objects of type **slice** have the **indices** function which takes the length of a sequence for which to calculate the indices:

```
>>> sl = slice(None, None, -1) # corresponds to [::-1]
>>> sl.indices(6)
(5, -1, -1)
>>> 'abcdef'[sl]
'fedcba'
```

The result, $(5, -1, -1)$, tells us that for sequences of the length 6, indices from 5 (last element) down to -1, but *not* including -1, will be used; in other words they will be 5, 4, 3, 2, 1, 0.

Similarly:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sl = slice(-2, 1, -2)
>>> sl.indices(9)
(7, 1, -2)          # 7 5 3
>>> a[sl]
[8, 6, 4]
>>> a[-2:1:-2]
[8, 6, 4]
```

Expressions like `[beg:end:step]` give us a copy of the (sub)sequence (in the reverse order for negative `step`). These constructs are often used, but there is a caveat we must be aware of. Consider:

```
>>> ls = [1, ['a', 'b'], 'c']
>>> copy = ls[::]
>>> reve = ls[::-1]
>>> reve[1][1] = 'd'
>>> copy
[1, ['a', 'd'], 'c']
>>> reve
['c', ['a', 'd'], 1]
>>> ls
[1, ['a', 'd'], 'c']
```

[1] Hence, `if ls==ls[::-1]:...` checks if ls is a palindrome.

Lists copy and reve contain *copies* of the second element of the original list ls (with index 1). However, this element is the *reference* to a list, and lists *are* mutable. We modify this list, (`reve[1][1] = 'd'`) using reve, but ls and copy still contain the reference to this, now modified, list. That's why all lists seem to be modified.

We can avoid this problem using a *deep* copy of a list, which copies references to immutable objects, for mutable elements creates (recursively, if needed) new objects. In the **copy** package, we have functions for both methods of copying: the program:

```
import copy
a = [1, ['a', 'b'], 2]
b = copy.copy(a)
c = copy.deepcopy(a)
print(id(a[0]), id(a[1]), id(a[2]))
print(id(b[0]), id(b[1]), id(b[2]))
print(id(c[0]), id(c[1]), id(c[2]))
```

printed

```
139948167725296 139948166858368 139948167725328
139948167725296 139948166858368 139948167725328
139948167725296 139948166871680 139948167725328
```

As we can see, references to numbers were just copied, as numbers are immutable. But for the second element of the original list, which is the reference to a list, the **copy** function also copied the reference, but **deepcopy** created new list, independent of the list pointed to by the same reference a[1] and b[1].

Slices can also be used to modify mutable sequences, like lists. We can remove subsequences defined by slices or replace them with something else:

```
>>> a = [1, 4, '5', [6, '7'], 2.25]
>>> a[2:4] = 'hello'                        # 1
>>> a
[1, 4, 'h', 'e', 'l', 'l', 'o', 2.25]
>>> a[1:3] = [8, 9, 'X']                    # 2
>>> a
[1, 8, 9, 'X', 'e', 'l', 'l', 'o', 2.25]
>>> a[2:-2] = ''                            # 3
>>> a
[1, 8, 'o', 2.25]
>>> a[1:3] = ['hello']                      # 4
>>> a
[1, 'hello', 2.25]
```

Note, that if the expression on the right is itself a sequence, it will be flattened and inserted into the sequence element by element (#1 and #2). If what we want is to insert a sequence as a single element, we can use one-element list containing this sequence (#4). Replacing a subsequence with an empty sequence just removes it (#3).

4.6 Call operator

Call operator is denoted by round parentheses and is used mostly to call (invoke) a function. But, as in C++, we can define classes, objects of which behave as if they were functions: we just provide the **operator()** method in C++, while in Python we define a special[1] **__call__** method — see sec. 11.8, p. 234. Also the object representing a class itself is callable — "calling" a class gives an object of this class (which itself may be callable, if the class defines the **__call__** method).

There is a built-in function **callable** which returns **True** if, and only if an object is callable:

```
>>> class Hello:
...     def __call__(self):
...         return 'Hello'
...
>>> h = Hello()
>>> print(h(), callable(Hello), callable(h))
Hello True True
```

4.7 *await expression*

The **await** expression is used in concurrent programming leveraging the ***asyncio*** package — see sec. **??**, p. **??**.

4.8 *Exponentiation*

Exponentiation (raising to a power, like x^y in mathematics) is denoted by double asterisk (**), as in Fortran, so what is x^y in math, would be `x**y` in Python. Its precedence is higher than that of unary operator on its left but lower than unary operator (−, in particular) on its right: `a**-2` is legal and interpreted as `a**(-2)`, while `-a**n` means `-(a**n)`.

The ** operator can be used where **pow** function is used in C/C++/Java:

```
>>> 2**8
256
>>> 2**-8
0.00390625
>>> 2**(-8)
0.00390625
>>> 8**(-1/3)
0.5
>>> 8**(1/3)
2.0
>>> 8**1/3      # this is just 8/3
2.6666666666666665
>>> 26**10      # number of ten-letter strings
141167095653376
>>> 33**33
129110040087761027839616029934664535539337183380513
```

Remember, though, that ^ *is not* exponentiation, as it is in some other languages: this symbol, as in C/C++/Java, is reserved for bitwise XOR operator!

The **pow** function exists also in Python (and is one of the builtin functions, so using it doesn't require any imports). The syntax is the same as in C/C++/Java, but there

[1] These methods are called "magic", or "dunder" methods.

is also a three-argument version: all three arguments must be then **int**s and the result is the remainder modulo third argument of the first argument raised to the power specified by the second: `pow(x, y, n)` is therefore $x^y \mod n$. For example

```
print(pow(2, 3), pow(2, 3, 5))
```

prints `8 3`

To define `**` operator for our classes, we have to implement the magic method __pow__ (see sec. 11.8, p. 234). For example:

```
class bizzarePow:
    def __init__(self, s):
        self.s = s
    def __pow__(self, p):
        return (self.s*p + '\n')*p*len(self.s)

print(bizzarePow('ab')**3)
print(bizzarePow('X')**5)
```

prints

```
ababab
ababab
ababab
ababab
ababab
ababab

XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

4.9 Unary operators

There are three unary operators in Python: Two of them

- numeric negation (sign reversal): `-` and
- unary plus, `+`, which applied to any numeric argument returns it unchanged

do not require any explanations, and the third, bitwise negation, `~`, will be covered in the section on bitwise operators (see sec. 4.11, p. 53).

4.10 Arithmetic operators

Binary arithmetic operators are generally known to us from C/C++/Java, although there are two specific for Python, and there are some differences in interpretation of some of them.

Five of them are of equal precedence - these are

- `*` — multiplication (no problems with that one);

- `@` — matrix multiplication: no standard Python types support it, but it's used in third-party packages, notably for scientific computing (like **NumPy**[1]);
- `/` — float division (but see below);
- `//` — floor division (see below);
- `%` — remainder (modulo) operator (see below).

The two remaining arithmetic operators have lower precedence than the five mentioned above, but are rather trivial

- `+` — addition;
- `-` — subtraction;

and no surprises here.

Let's talk only about non-trivial operators, whose interpretation is not always so obvious.

4.10.1 Division operators

The `/` operator corresponds to division. If at least one of the operands is `float` (which is `double` in C/C++/Java), the result is also `float` and no surprises here.

However, in C/C++/Java, this operator applied to integers yields integer, truncating the result towards zero. In Python, even if both operands are `int`s, the result is of type `float`!

So, in Java

```
jshell> 9/3     // JAVA
$1 ==> 3

jshell> 9/4
$2 ==> 2

jshell> 1/3
$3 ==> 0

jshell> -4/3
$4 ==> -1
```

but in Python

```
>>> 9/3         #  PYTHON
3.0
>>> 9/4
2.25
>>> 1/3
0.3333333333333333
>>> -4/3
-1.3333333333333333
```

For integer division in Python, we have a special operator `//`. Therefore, if we divide two integers using this operator, the result will be truncated but *not*, as in C/C++/Java, towards zero, but towards $-\infty$, downwards (and hence it's called *floor division*). In Java, we have

[1] https://numpy.org

```
jshell> 5/4     // JAVA
$1 ==> 1

jshell> -5/4
$2 ==> -1       // -1.25 truncated towards zero

jshell> Math.floor(-5./4)
$3 ==> -2.0
```

while in Python

```
>>> import math #  PYTHON
>>> 5//4
1
>>> -5//4       #  -1.25 truncated towards minus infinity
-2
>>> math.floor(-5/4)
-2
```

so // is like float division / followed by taking the floor of the result.

4.10.2 Modulo operator

As in C/C++/Java, modulo , or remainder, operator % does not correspond to the modulo operator in mathematics for negative operands. But the behavior of this operator in Python is different from that in C/C++/Java.

In all these languages, the following formula must be true for all integer values of a and b (with $b \neq 0$, of course)

$$(a/b) * b + a\%b = a$$

where integer division is understood by /.

Using a small program in C++

```
#include <iostream>
#include <cstdio>
void pr(int a, int b) {
    static int fst = 0;
    if (++fst == 1)
        std::printf("C++\n  a   b a/b a%%b (a/b)*b+a%%b\n");
    std::printf("%3d%4d%3d%4d%9d\n",a,b,a/b,a%b,(a/b)*b+a%b);
}
int main() {
    pr( 17,  10);
    pr( 17, -10);
    pr(-17,  10);
    pr(-17, -10);
}
```

which prints

```
C++
  a   b a/b a%b (a/b)*b+a%b
 17  10  1   7       17
 17 -10 -1   7       17
```

```
-17  10 -1  -7      -17
-17 -10  1  -7      -17
```

we can verify that indeed $(a/b) * b + a\%b = a$. The rule for negative operands is simple here: apply modulo operator to absolute values of the operands and then take the result with the sign of a (ignoring the sign of b).

However, for this formula to hold, the definition of the modulo operator in Python must be different than that in C/C++/Java, because the definition of integer division is different!

A similar program in Python

```
def pr(a, b, fst=[]):
    if not fst:
        print("PYTHON\n  a   b a//b a%b (a//b)*b+a%b")
        fst.append(1)
    print(f"{a:3}{b:4}{a//b:4}{a%b:4}{(a//b)*b+a%b:9}");

pr( 17,  10);
pr( 17, -10);
pr(-17,  10);
pr(-17, -10);
```

prints

```
PYTHON
  a   b a//b a%b (a//b)*b+a%b
 17  10   1   7       17
 17 -10  -2  -3       17
-17  10  -2   3      -17
-17 -10   1  -7      -17
```

and the easiest way to grok the result with negative arguments is to remember that $a\%b = a - (a/b) * b$ — this works for both Python and C/C++/Java with / assumed to be the integer division appropriate for the language at hand (// in Python).

As we can see, the best we can do is to avoid using integer divisions and modulo operators if we suspect that operands might be negative.

4.11 Bitwise operators

There are six bitwise operators in Python; in the order of descending precedence, these are:

- bitwise NOT (~),
- left (<<) and right (>>) shifts,
- bitwise AND (&),
- bitwise XOR (^),
- bitwise OR (|).

In Python, bitwise operators are sometimes tricky. The problem with them originates from the fact that integer values are represented in a complicated way (see sec. 3.4.1, p. 33). This representation doesn't have anything to do with integers in C/C++/Java. In these languages, integer values are represented always by a fixed-length sequence of bits, with the most significant bit, the so called sign bit, treated specially (for *signed* types). Because the length (number of bits) is known, it is also known which bit is the most significant one. However, representation of integers in Python is completely different and by construction allows to represent arbitrary large numbers, whose "normal" two's complement binary representation would require arbitrarily long sequences of bits. As the first (from the left) non-zero bit is always 1, it would be not known if contribution from this bit is to be taken with plus sign (positive number) or with minus sign (negative numbers).

Shift operators work as expected, but with some quirks. Shifting to the left is *always* equivalent to multiplication by a power of 2:

```
m << n   ≡   m * pow(2, n)
```

For positive m, we can easily check that this indeed corresponds to shifting bits of "normal" binary representation of m by n bits to the left, with zeros entering from the right (below we use **bin** function to display binary representation of a *positive* number):

```
>>> a = 153
>>> bin(a)
'0b10011001'
>>> b = a << 3
>>> bin(b)
'0b10011001000'
>>> bin(a*2**3)
'0b10011001000'
```

Note that in a fixed-length representation, as in C/C++/Java, leftmost bits may be dropped if they are shifted beyond the left "edge" of the binary representation of the number: then shifting will *not* be equivalent to multiplication by a power of 2, and the result may even become negative (for signed types): the following snippet in C++

```
int a = 100'000'000;
cout << (a << 10) << endl;
```

prints

```
-679215104
```

and similarly, in Java

```
jshell> int a = 100_000_000
a ==> 100000000

jshell> System.out.println(a << 10)
-679215104
```

In Python, however, it's not a problem, as the length of the binary representation of a number is not limited:

```
>>> a = 1_000_000
>>> b = a << 10
>>> c = a * 2**10
>>> bin(a)
'0b11110100001001000000'
```

```
>>> bin(b)
'0b111101000010010000000000000000'
>>> bin(c)
'0b111101000010010000000000000000'
>>> b, c
(1024000000, 1024000000)
>>> b == c
True
```

With negative numbers the situation is a little worse. The equivalence

```
m << n   ≡   m * pow(2, n)
```

still holds. However, it's not so simple to see the bit pattern, because for negative numbers, the **bin** function shows the bit representation of the absolute value with minus sign added at the front, not the two's complement bit representation of the number itself:

```
>>> a = 913
>>> b = -a
>>> bin(a)
'0b1110010001'
>>> bin(b)
'-0b1110010001'
```

However, we can use the **to_bytes** method to see individual bytes:[1]

```
>>> a = 913
>>> b = -a
>>> c = b << 2
>>> d = b * 2**2
>>> c == d
True
>>> c, d
(-3652, -3652)
>>> e = d.to_bytes(4, 'big', signed=True) # 4 bytes as int in Java
>>> e
b'\xff\xff\xf1\xbc'
>>> int.from_bytes(e, 'big', signed=True)
-3652
```

`\xff\xff\xf1\xbc` is 11111111111111111111000110111100 in binary system and indeed, we can check in Java that this *is* -3652:

```
jshell> a = -3652
a ==> -3652

jshell> Integer.toBinaryString(a)
$1 ==> "11111111111111111111000110111100"
```

[1] Calling this method, we specify that the result is to be on 4 bytes (as **int** in C/C++/Java) and we want to see bytes ordered from the most to the least significant. [Instead of little or big, we can use sys.byteorder which gives the byte order on the current platform; usually little.]

Bitwise AND (`&`), bitwise OR (`|`) and bitwise XOR (`^`) work as expected (again, it's better not to touch negative numbers):

```
>>> m = 0b11001010
>>> n = 0b11010
>>> bin(m & n)
'0b1010'
>>> bin(m | n)
'0b11011010'
>>> bin(m ^ n)
'0b11010000'
```

Bitwise NOT (bit-flip, `~`) is not so trivial. As integers are unlimited, we can imagine that their representation for a positive number contains an infinite sequence of zeros to the left of the most significant bit which is set. Flipping the bits would then create an infinite sequence of ones. And this is what happens: of course, all these ones are not physically present, but conceptually they are there.

Let us show what's going on when we bitwise negate a number. Consider, for example, 321_{10} $(=101000001_2)$. To represent it, 9 bits are needed

```
>>> bin(321)
'0b101000001'
>>> (321).bit_length()
9
```

Python adds one leading zero (as it is positive), takes the sequence of 10 bits and negates them:

```
0101000001          # 321 plus leading 0
1010111110          # flipped
```

and this is interpreted as two's complement representation of a 10-bit integer; the first bit is treated as the sign bit, so its contribution will be negative and the number is $-2^9 + 2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$ or $-512 + 128 + 32 + 16 + 8 + 4 + 2 = -322$.

```
>>> a = 321
>>> nega = ~a
>>> bin(a)
'0b101000001'
>>> bin(nega)
'-0b101000010'
>>> a, nega
(321, -322)
>>> negabytes = nega.to_bytes(2, 'big', signed=True)
>>> negabytes
b'\xfe\xbe'
>>> int.from_bytes(negabytes, 'big', signed=True)
-322
```

`\xfe\xbe` is 1111111010111110: there is "infinite" sequence of ones at the beginning, so the number is negative and only the last one is significant (giving, as we showed above, -322).

As we remember, in two's complement notation, reversing the sign is equivalent to flipping all bits followed by addition of 1. We can check, that this feature is preserved:

```
>>> ~321+1
-321
>>> ~-9987654321+1
9987654321
>>> ~123123123123123123123+1
-123123123123123123123
>>> ~99999999999999999999999999999999999999999999999999+1
-99999999999999999999999999999999999999999999999999
```

4.12 Comparisons

4.12.1 Arithmetic comparisons

There are six arithmetic comparison operators comparing *values* of objects, which are well known from many other languages: <, < =, ==, !=, > = and >. There is no point in explaining them, but one of their features is specific to Python — there are separate rules for chains of these operators and these rules cannot be expressed as associativity (right or left). Such chains would be illegal in Java. In C/C++, they would compile but give nonsensical results; for example

```
int a = 1, c = 3;
cout << boolalpha << (c < a < a) << endl;
```

prints **true**.

Such chains of (in)equalities *are* legal and make sense in Python, though. For example,

```
a < b < c
```

is neither

```
(a < b) < c
```

(as it would be in C/C++), nor

```
a < (b < c)
```

but

```
a < b and b < c
```

(but b will be evaluated once only). Chains of (in)equalities is equivalent to ANDed chains of separate binary operations evaluated form left to right:

```
>>> 5 < 7 > 2
True
>>> 5 < 7 and 7 > 2
True
>>>
>>> 5 < 7 > 7
False
>>> 5 < 7 and 7 > 7
False
>>>
>>> a = 5; b = 7
>>> a < b == b - a + 5
True
>>> a < b and b == b - a + 5
True
>>>
>>> a == b - 2 < b >= a
```

```
True
>>> a == b - 2 and b - 2  < b and b >= a
True
```

4.12.2 Identity comparisons

There are two identity comparison operators: **is** and **is not**. They are (rather seldom) used to check if two objects are actually the same object, or two distinct objects, perhaps representing the same value. Remember, that == is supposed to compare values, not IDs (addresses). It is defined for many Python builtin types. But, if it's not defined — e.g., for user defined types — the default implementation, defined in class **object**, will be used, which is equivalent to **is** — comparing addresses (this is analogous to the **equals** method in Java).

For type **str** (string), the == operator is, of course, well defined (as is **String::equals** in Java):

```
>>> a = 'Hello, '; b = 'World'
>>> s1 = a + b
>>> s2 = 'Hello, World'
>>> s1 is s2
False
>>> s1 == s2
True
>>> s1 is not s2
True
```

We remember, that small integers are created once only and all references to the same small value refer to exactly the same object (see sec. 3.4.1, p. 33): we can check it using the **is** operator

```
>>> a = 17; b = 17
>>> a == b
True
>>> a is b
True
>>> id(a), id(b)
(9801760, 9801760)
>>>
>>> # but:
>>>
>>> a = 1200; b = -1000; c = a - b - 1000
>>> a == c
True
>>> a is c
False
>>> id(a), id(c)
(140120910778288, 140120910340336)
```

4.12.3 Membership test operators

Membership tests belong to the same precedence group as comparisons. Operators **in** and its negation **not in** check if an object is a member of a collection: list, tuple, dictionary, set, but also string, which is a sequence of individual characters. For dictionaries, a key is searched for, values are ignored. The usage of these operators is straightforward:

```
ls = [(1, 2), (3, 4, 5), (6, 7, 8, 9)]
di = {'one': 1, 'two':2, 'three':3}
se = {1, 2, 3, 4}
st = 'abcde'
print((1, 2) in ls)    # True
print((2, 1) in ls)    # False
print(4 in ls[1])      # True
print(4 in ls[2])      # False
print('one' in di)     # True
print(2 in di)         # False
print(4 in se)         # True
print(5 in se)         # False
print('cd' in st)      # True
print('f' in st)       # False
```

4.13 Boolean not, and and or operators

Boolean operators: **not**, **and** and **or** correspond to operators known from C/C++/Java: !, && and ||, respectively.

As in these languages, **and** and **or** operators are short-circuited — the second operand is not evaluated if the result is known after evaluation of the first.

As we remember (see sec. 3.4.2, p. 35) values of any type may be used in a context requiring a Boolean: numerical values equal to zero, empty sequences and collections, None and False are all "falsy", everything else is "truthy".

It follows, that operands of Boolean operators can be of any type, not necessarily bool — they will be treated as True or False anyway. But what about the type of a return value?

For the **not** operator, the return value is True or False:

```
>>> not {}
True
>>> not {7}
False
>>> not 3
False
>>> not 0
True
>>> not [1, 2, 3]
False
>>> not []
True
>>> not 'a'
False
>>> not ''
True
>>> not False
True
```

```
>>> not True
False
```

However, the **and** and **or** operators do not return **True** or **False**, but simply one of their operand (which, however, always can be interpreted as **True** or **False** in the logical context) according to the following rules:

- Conjunction: `expr1 and expr2` returns
  - `expr1` (and `expr2` is not even evaluated) if `expr1` is “falsy”;
  - `expr2` otherwise.
- Alternative: `expr1 or expr2` returns
  - `expr1` (and `expr2` is not even evaluated) if `expr1` is “truthy”;
  - `expr2` otherwise.

For example:

```
>>> a = 0 and [1, 2]
>>> a, type(a), bool(a)
(0, <class 'int'>, False)
>>>
>>> a = -1 and [1, 2]
>>> a, type(a), bool(a)
([1, 2], <class 'list'>, True)
>>>
>>> a = -1 and {}
>>> a, type(a), bool(a)
({}, <class 'dict'>, False)
>>>
>>> a = [] or {}
>>> a, type(a), bool(a)
({}, <class 'dict'>, False)
>>>
>>> a = [] or {1, 2}
>>> a, type(a), bool(a)
({1, 2}, <class 'set'>, True)
```

As this example shows, the **bool** function — strictly speaking, a constructor of the **bool** class — can be used to convert any value to a “true” Boolean.

4.14 Conditional expression

Conditional expression corresponds to ternary operator **?:** in C/C++/Java. The syntax is a little different, though:

```
expr1 if condition else expr2
```

and the value of the whole expression is the value of

- expr1 if `condition` evaluates to **True** (and then expr2 is not evaluated at all);
- expr2 if `condition` evaluates to **False** (and then expr1 is not evaluated).

Note that it is acceptable if the value of expr1 and/or expr2 is **None** — for example, if it is a function call and the function doesn’t return anything (i.e., returns **None**):

```
a = 6
b = print('even') if a % 2 == 0 else print('odd')
print(b)
```

prints

```
even
None
```

Another example:

```
n = p = 7
while p > 1:
    print(n, end=' ')
    p, n = n, n//2 if n%2 == 0 else 3*n + 1
```

prints the Collatz (hailstone) sequence starting from 7

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

The following program checks which of the Fibonacci numbers F_n for $n \in [3, 15]$ are prime; here, we use not only conditional expression, but also, in order to avoid recalculating Fibonacci number, assignment expression (see the last line):

Listing 15 ABB-Cond/Cond.py

```
#!/usr/bin/env python

def is_prime(n):
    if n <= 1: raise ValueError
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    k = 5
    while k**2 <= n:
        if n % k == 0 or n % (k+2) == 0: return False
        k += 6
    return True

def fib(n):
    f = 0; s = 1
    if n < 2: return n
    for i in range(n-1): f, s = s, f+s
    return s

for n in range(3, 16):
    print(f"{v} yes" if is_prime(v := fib(n)) else f"{v} no")
```

The program prints

```
2 yes
3 yes
5 yes
```

```
8 no
13 yes
21 no
34 no
55 no
89 yes
144 no
233 yes
377 no
610 no
```

Conditional expressions may be nested

```
expr1 if condition1 else expr2 if condition2 else expr3
```

Using such a vague form is discouraged, as it's hard to read and understand. Conditional expressions are right-associative (as for ?: in C/C++/Java), so the above is equivalent to

```
expr1 if condition1 else (expr2 if condition2 else expr3)
```

which is easier to comprehend; the value of the whole expression is therefore

- expr1 if condition1 is true — the remaining conditions and expressions are not evaluated then;
- expr2 if condition1 is false, but condition2 is true;
- expr3 if both condition1 and condition2 are false.

For example:

```
a = 1; b = 7
x1 = 'expr1' if a < b else 'expr2' if a > b else 'expr3'
x2 = 'expr1' if a > b else 'expr2' if a < 2 else 'expr3'
x3 = 'expr1' if a > b else 'expr2' if b < a else 'expr3'
print(x1)
print(x2)
print(x3)
```

prints

```
expr1
expr2
expr3
```

Conditional expressions can be used in list comprehensions; for example here we replace all numbers from range $[-1, 8)$ that are smaller than 2 by 2 and those greater than 5 by 5:

```
ls = [5 if n > 5 else 2 if n < 2 else n for n in range(-1, 8)]
print(ls)  # prints [2, 2, 2, 2, 3, 4, 5, 5, 5]
```

Chapter 5

Functions

5.1 *Defining functions. Scopes of variables*

In Python, functions are represented, as everything else, by objects. A simple definition of a function may look like this

```
def mul(x, y):
    return x*y

def printData(x):
    print('Data is {}'.format(x))  # No return
```

As we can see, the return type is *not* declared. Actually, all Python functions return something — even if a function ends with just **return,** (without specifying any value to be returned) or does not contain any **return** statements at all — it then returns **None**.

Also, parameters types remain unspecified: their type will be determined dynamically by the type of arguments passed to the function every time it is being invoked. This means, that the same function may be called with arguments of completely different and unrelated types and, if the syntax of the body of the functions turns out to be correct for these types, all such invocations will be correct — this feature is called *duck typing*.

```
>>> def mul(x, y):
...     return x*y
...
>>> def printData(x):
...     print('Data:', x, type(x))
...
>>> printData(1.5)
Data: 1.5 <class 'float'>
>>>
>>> printData((1, 2, 3))
Data: (1, 2, 3) <class 'tuple'>
>>>
>>> printData([1, 'Eve'])
Data: [1, 'Eve'] <class 'list'>
>>>
>>> printData(r := mul(3, 3))
Data: 9 <class 'int'>
>>>
>>> printData(r := mul((1+2j), (2-4j)))
Data: (10+0j) <class 'complex'>
>>>
>>> printData(r := mul([1, 2], 3))
Data: [1, 2, 1, 2, 1, 2] <class 'list'>
```

Contrary to most other languages, definition of a function is an executable statement. For example, the definition of the **printData** function above is syntactically similar to just defining a numeric variable `x = 7`. The name printData is, as usual, the name of a *reference* which points to the object representing our function. We can create other references to the same object:

```
>>> pp = printData
>>> pp(1+2j)
Data: (1+2j) <class 'complex'>
>>> printData = 17
>>> printData
17
```

However, the original name of a function is kept as one of the attributes of the object representing it, but this can be easily changed:

```
>>> dir(pp)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']
>>> pp.__name__
'printData'
>>> pp.__name__ = 'pp'
>>> pp.__name__
'pp'
```

Functions, being objects, can be created inside other functions, passed as arguments to and/or returned from other functions. Functions returning or taking as their argument(s) other functions are called **higher order functions**.

In the example below, the **compos** function takes two functions, f and g, and returns also a function, namely their composition[1]

Listing 16 AAP-FunCompos/Compos.py

```python
#!/usr/bin/env python

def compos(f, g):
    def _h(x):
        return f(g(x))
    # just for fun...
    _h.__name__ = g.__name__ + '_and_then_' + f.__name__
    return _h

def squa(x):
    return x*x
def add2(x):
    return x+2
```

[1] The composition of functions f i g is a function $f \circ g$ such that $(f \circ g)(x) \equiv f(g(x))$.

```python

sqad = compos(squa, add2)
adsq = compos(add2, squa)

print(type(squa), type(add2))

print('name of sqad:', sqad.__name__)
print('name of adsq:', adsq.__name__)

print(sqad(2))    # 16
print(adsq(2))    #  6
```

The program prints

```
<class 'function'> <class 'function'>
name of sqad: add2_and_then_squa
name of adsq: squa_and_then_add2
16
6
```

Let us look at another example:

Listing 17 AAR-FunCount/Counter.py

```python
 1  #!/usr/bin/env python3
 2
 3  def getCounter():
 4      n = 0
 5      def count():
 6          nonlocal n
 7          n += 1
 8          return n
 9      return count
10
11  f = getCounter()
12  for i in range(3):
13      print(f(), end = ' ')   # 1 2 3
14
15  print('\n' + str(type(f)))  # <class 'function'>
16  print(f.__name__)           # count
```

What is interesting here is the **nonlocal** keyword (line 6). This leads us to the notion of a **scope**: a part of a program in which a set of name bindings is visible and directly (just by a single, non-qualified name) accessible. At run-time, names are searched for in the following scopes:

1. Innermost scope with **L**ocal names.
2. The scope of **E**nclosing functions, if any (applicable for nested functions).
3. The current module's **G**lobal scope.
4. The scope of the **B**uilt-in entities.

When a name is used, its meaning is searched for in these scopes in the order as above — this is known as the **LEGB** rule. There are some principles governing the search:

- in a local scope (inside a function), the name appearing on the left-hand side of an assignment is considered to be the name of a local variable (which is created, if it doesn't exist);
- otherwise, a name refers to a local variable, if it already exists; if not, the surrounding non-global scope (if there is any) is examined, then the global scope of the module, and finally the built-in namespace;
- if a global variable is to be assigned a new value inside a local scope (in a function, for example), it has to be declared as **global** before it is used.
- Suppose we have an inner scope inside another local scope (like in definition of a function inside another, surrounding function). Then, in order to assign a new value to a variable from the surrounding scope (but *not* the global one) inside the inner scope, it has to be declared in the inner scope as **nonlocal** before being used.

For example, the code below will not work: `n` inside the function is assigned a value, so it is considered to be a new local variable. However, the value which is to be assigned depends itself on the value of `n`:

```
>>> n = 1
>>> def fun():
...     n = n + 7
...
>>> fun()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fun
UnboundLocalError: local variable 'n' referenced before assignment
```

but this will work

```
>>> n = 1
>>> def fun():
...     global n
...     n = n + 7
...
>>> fun()
>>> print(n)
8
```

Similarly, this is incorrect

```
>>> def fun():
...     n = 9
...     def inn():
...         n += n
...         return n - 8
...     return inn
...
>>> f = fun()
>>> print(f())
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in inn
UnboundLocalError: local variable 'n' referenced before assignment
```

but the code below is OK: as we don't attempt to assign a new value to `n`, it will be searched for, and found, in the surrounding scope)

```
>>> def fun():
...     n = 9
...     def inn():
...         return n - 8
...     return inn
...
>>> f = fun()   # f is now inn
>>> print(f())
1
```

The code below is correct as well, as is this version, where we declared `n` as **nonlocal**

```
>>> def fun():
...     n = 9
...     def inn():
...         nonlocal n
...         n += n
...         return n - 8
...     return inn
>>> f = fun()
>>> print(f())
10
```

And another example. The variable `n` is created when the **halves** function is called (one time only). Inside the nested function **addhalf**, it's declared as **nonlocal**, so it is the same variable defined on line 7. Note that it is used (and redefined) inside the **addhalf** function which is returned from **halves**. One could think that after the return, the variable `n` doesn't exist any more, as it was local in the **halves** function. However, it will be copied do the so called **closure** of **addhalf**, and can be used and modified by this function:

Listing 18 AAS-Scopes/Scopes.py

```
1  #!/usr/bin/env python
2
3  n = 3.0                              # global
4  print('Global n:', id(n), n, end = '\n\n')
5
6  def halves():
7      n = -0.5                         # local
8      print('Local  n:', id(n), '\n')
9      def addhalf():
10         nonlocal n                   # n from line 7
11         print('Inner1 n:', id(n), n) # same as local
12         n += 0.5
```

```
        print('Inner2 n:', id(n), n) # changed now
        return n
    return addhalf

f = halves()
for i in range(2):
    print(f(), end = '\n\n')

print('Global n:', id(n), n)
print('From closure:',f.__closure__[0].cell_contents)
```

The output of the program is

```
Global n: 139640753183728 3.0

Local  n: 139640753183472

Inner1 n: 139640753183472 -0.5
Inner2 n: 139640753183632 0.0
0.0

Inner1 n: 139640753183632 0.0
Inner2 n: 139640753183600 0.5
0.5

Global n: 139640753183728 3.0
From closure: 0.5
```

[See also discussion in section on lambdas, sec. 5.3, p. 74]

Contrary to languages like C/C++/Java, blocks of code (for example under **if** of inside a loop) do *not* create new scopes. Variables defined in such blocks are just dynamically added to the dictionary of local names and will stay there even when the flow of control leaves the block. Consider the following example

```
x = int(input('Enter x (negative or positive) '))

if x >= 0:
    a = 'a'
else:
    b = 'b'

print('"a" exists?', 'a' in locals())
print('"b" exists?', 'b' in locals())
```

If, running this program, we enter a positive number, variable `a` will exist after the if statement, but `b` will not

```
Enter x (negative or positive) 4
"a" exists? True
"b" exists? False
```

while with a negative number as input

```
Enter x (negative or positive) -3
"a" exists? False
"b" exists? True
```

`b` will exist, but `a` will not.

5.2 Passing arguments

Let's look at the following example

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> fun(1, 'Hello', 2.5)
"a=1 b='Hello' c=2.5"
>>>
>>> fun(c='World', a=17, b=(3, '3'))
"a=17 b=(3, '3') c='World'"
>>>
>>> fun(21, c=8.28, b='Ottawa')
"a=21 b='Ottawa' c=8.28"
>>>
>>> fun(a=21, 'Ottawa', c=8.28)
  File "<stdin>", line 1
    fun(a=21, 'Ottawa', c=8.28)
                              ^
SyntaxError: positional argument follows keyword argument
```

Parameters of **`fun`** are *positional* — there are exactly three of them and they don't have any default values. Therefore, when invoking the function, we have to pass exactly three arguments. However, as we can see from the second invocation, their order can be changed, if *names* of parameters are specified (these are called **keyword arguments**).

In the third invocation, the name of the first argument hasn't been specified, so it must correspond to the first parameter. Generally, *unnamed* arguments must be given first and in the correct order; the remaining arguments may be passed to the function in any order as keyword arguments. However, if an argument was passed by keyword, all the remaining arguments must also be passed by keyword; otherwise we'll get an exception, as in the last invocation above.

It may happen that a function expects several arguments, but in our program we have them collected into a sequence (a list or a tuple). We cannot pass just a list (or a tuple) because that would be *one* argument (of type **`list`** or **`tuple`**). However, when passing a collection, we can "unpack" it into a sequence of its single elements separately using the `*`-operator:

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> lst = ['a', 1, 'Hello']
>>> fun(*lst)
"a='a' b=1 c='Hello'"
```

```
>>> tpl = 6, 'World', 1.5
>>> fun(*tpl)
"a=6 b='World' c=1.5"
```

In a similar way, we can unpack a dictionary into a sequence of keyword arguments using the **-operator:

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> dct = dict(b = 'Hello', c = 1.25)
>>> fun('First', **dct)
"a='First' b='Hello' c=1.25"
```

Here, the first argument was not a keyword one, so it has to correspond to the first positional parameter; the remaining arguments will be keyword arguments after unpacking the dictionary.

Another example:

```
#!/usr/bin/env python

def fun(a, b, c, d, e, f):
    print(a, b, c, d, e, f)

fun(1, *(2, 3), f=6, *(4, 5))
fun(*(1, 2), e=5, *(3, 4), f=6)
fun(1, **{'b': 2, 'c': 3}, d=4, **{'e': 5, 'f': 6})
fun(c=3, *(1, 2), **{'d': 4}, e=5, **{'f': 6})
```

prints

```
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

Some or all arguments of a function can have default values (as in C++, but not Java or C). If there are parameters with default values, they have to be listed *after* parameters without them. Functions with default arguments can be called with fewer arguments — the interpreter will use the default values for arguments' values that are missing. Let's look at the example:

Listing 19 AAU-DefArgs/DefArgs.py

```
#!/usr/bin/env python3

def f(a, b='defaultB', c=None):
    if c == None:
        print("c hasn't been specified!", end = ' ')
    print(f'{a=} {b=} {c=}')
f(1, 2, 3)  # a=1 b=2 c=3
```

```
 8  f(1, 2)     # c hasn't been specified! a=1 b=2 c=None
 9  f(1)        # c hasn't been specified! a=1 b='defaultB' c=None
10  f(c='c', a='a', b='b')  # a='a' b='b' c='c'
11
12  i = 7
13  def fundef(a, b=i): # default value of b will always be 7!
14      return a + b;
15
16  i += 10
17    # this will print 'i = 17 fundef(1) = 8'
18  print('i =', i, 'fundef(1) =', fundef(1))
19
20    # important example
21  def funlist(val, lst = []): # default list created once!
22      lst.append(val)
23      return lst;
24  funlist(1)
25  funlist(2)
26  funlist(3)
27  print(funlist(4))                # [1, 2, 3, 4]
28  print(funlist('c', ['a', 'b'])) # ['a', 'b', 'c']
29  print(funlist('d'))              # [1, 2, 3, 4, 'd'] !
```

The example illustrates a very important problem that we have to keep in mind when using default values for arguments of functions. The object representing a default value is created once only: when the interpreter evaluates the definition of a function. In all subsequent invocation, reference to exactly the same object containing the default value will be used. For example, in line 13, we define a function with default value. This default value will be 7 “forever”. Even if we change variable i on line 16, and invoke the function again, the old value of the default value, i.e., 7, will be used. This is clear when the default object is immutable (a number, a strings, ...). However, as we can see from the example with empty list as the default value (line 21), it can lead to surprises if this object *is* mutable (like, e.g., a **list**). Then in all invocations, when the corresponding argument is omitted, the same object is used and its value will be as it was left after the previous invocation with this argument omitted. Consequently, in all subsequent invocations, the list will *not* be empty!

Python supports variadic functions, i.e., functions with arbitrary number of arguments. A sequence of arguments (perhaps empty) is denoted, on the parameter list, by *args (the name args is conventional, what matters is the asterisk). When invoking the function, all arguments (after the obligatory, positional arguments, but before keyword arguments, if any) will be collected into a tuple and passed to the function — therefore, inside the function, args is the reference to a, perhaps empty, tuple:

```
 #!/usr/bin/env python

def f(a, *args):
    print("Positional:", a, end=', ')
    if len(args) == 0:
        print("No var args")
        return
    print("Var args:", end=' ')
    for i, v in enumerate(args):
        print(f'{i+1} -> {v};', end=' ')
    print()

f(1)
f(2, 'A', 1.5, 'B')
f(3, ('X', 'Y'))
f(4, *('X', 'Y'))
```

The program prints

```
Positional: 1, No var args
Positional: 2, Var args: 1 -> A; 2 -> 1.5; 3 -> B;
Positional: 3, Var args: 1 -> ('X', 'Y');
Positional: 4, Var args: 1 -> X; 2 -> Y;
```

Note the last two invocations: in the third we pass tuple `('X', 'Y')` as a single argument, while in the fourth, we unpack the tuple into two individual arguments, which will be gathered back to a tuple args on the function's side.

At the very end of the parameter list of a function, one can write **kwargs (again, the name kwargs is conventional, what matters is the double asterisk). When calling the function, we pass any number of keyword arguments (in the form `k1=v1, k2=v2,...`). They will be gathered into a dictionary and passed to the function — therefore, inside the function, kwargs is the reference to a, perhaps empty, **dict**:

Listing 20 AAW-ArgsKwargs/ArgsKwargs.py

```python
#!/usr/bin/env python

def f(a, *args, **kwargs):
    print('Positional:', a, end=', ')
    if len(args) == 0:
        print('No var args;', end=' ')
    else:
        print('Var args:', end=' ')
        for i, v in enumerate(args):
            print(f'{i+1} -> {v};', end=' ')
    if len(kwargs) == 0:
        print('No kwargs;')
    else:
        print('KWargs:', end=' ')
        for k, v in kwargs.items():
            print(f'{k} : {v}; ', end = ' ')
        print()

f(1)
f(2, 'x', 'y')
f(3, fName='Sue', lName='Lee')
f(4, 'x', 'y', w=24, h=30)

```

```
lst = ['a', 'b']
dic = dict(sue="S", joe="J")
f(5, *lst, **dic)
```

In the last invocation, we first collected our arguments into a list and a dictionary, and then we unpacked them when passing to the function: they are packed back by the interpreter to the args tuple and kwargs dictionary on the function's side.

The program prints:

```
Positional: 1, No var args; No kwargs;
Positional: 2, Var args: 1 -> x; 2 -> y; No kwargs;
Positional: 3, No var args; KWargs: fName : Sue;  lName : Lee;
Positional: 4, Var args: 1 -> x; 2 -> y; KWargs: w : 24;  h : 30;
Positional: 5, Var args: 1 -> a; 2 -> b; KWargs: sue : S;  joe : J;
```

It is also possible to force some parameters to be "positional-only" or "keyword-only". There is a "special" parameter: just single /. It means that arguments preceding this position on the parameter list *cannot* be given names:

```
>>> def fun(a, b, c, /, d):
...     print(a, b, c, d)
...
>>> fun(1, 2, 3, 4)
1 2 3 4
>>> fun(1, 2, 3, d=4)
1 2 3 4
>>> fun(1, 2, c=3, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() got some positional-only arguments
  passed as keyword arguments: 'c'
```

We can also force some arguments to be passed as keyword-only. One way of doing it is as follows

```
>>> def fun(*args, a):
...     print(args, a)
...
>>> fun(1, 2, 3, a=4)
(1, 2, 3) 4
>>> fun(a=4)
() 4
>>> fun(1, 2, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() missing 1 required keyword-only argument: 'a'
```

Here we have to specify the name of a — otherwise it would be "swallowed" by the args tuple.

There is also a "special" parameter denoted by *. All arguments after the position marked with this "parameter" *must* be passed by keyword

```
>>> def fun(a, b=2, *, c, d):
...     print(a, b, c, d)
...
>>> fun(1, c=3, d=4)
1 2 3 4
>>> fun(1, 2, c=3, d=4)
1 2 3 4
>>> fun(1, 2, 3, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() takes from 1 to 2 positional arguments but
3 positional arguments (and 1 keyword-only argument) were given
```

An example is provided by the built-in function **sorted** which is declared as

```
sorted(iterable, /, *, key=None, reverse=False)
```

which means that the first argument (an iterable) must be passed without any name, while using optional arguments (key and/or reverse) requires specifying their names

```
>>> lst = ['Jane', 'Sophia', 'Liz']
>>> sorted(lst, reverse=True)
['Sophia', 'Liz', 'Jane']
>>> sorted(lst, key=lambda e: e[-1])
['Sophia', 'Jane', 'Liz']
>>> sorted(lst, reverse=True, key=lambda e: e[-1])
['Liz', 'Jane', 'Sophia']
```

5.3 Lambdas

Another way of creating functions is by using a **lambda** expression. Despite the protests of Guido van Rossum, they were added to Python under pressure of users (and fans of the Lisp programming language).

The syntax of a lambda function is very simple:

```
lambda args_list: expr
```

where args_list is a list of comma separated list of formal parameters and expr is an expression (i.e., something that has a value) in which the parameters can be used. The value of expr will be returned by the function represented by this lambda; note, however, that there is no explicit **return** statement here.

As expr has to be *one expression*, one cannot, for example, use **if-else** statement; conditional expressions, however, can be used:

```
pal = lambda ls: 'palindrom' if ls == ls[::-1] else 'not palindrom'

print(pal('rotor')) # prints 'palindrom'
print(pal('rover')) # prints 'not palindrom'
```

The example above illustrates also the fact, that lambdas may be given names, which then can be used exactly as names of functions

```
q = lambda x: 0.25*x**2 - x - 3
print(q(2), q(3), q(-4))          # prints -4.0 -3.75 5.0
```

Lambda expressions' are of type **function** and they can be used as (often anonymous) functions created "on the fly", like lambdas in Java or C++. Note, however, that they are not so useful as they are in Java or, even more so, in C++, because the body of a lambda-function is limited to a single expression the value of which is returned.

Any lambda is basically equivalent to a function, as **sq** and **lam** below

```
>>> def sq(x):
...     return x*x
...
>>> lam = lambda x: x*x
>>> type(sq), type(lam)
(<class 'function'>, <class 'function'>)
>>>
>>> dir(sq)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>>
>>> dir(lam)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>>
>>> sq.__name__, lam.__name__
('sq', '<lambda>')
```

As we see, these two objects are essentially the same, except for their `__name__` attribute. However, lambdas can be used in contexts where one couldn't use a definition of a function — for example directly on the argument list of a function invocation, as in the example below:

```
>>> def apply_n_times(f, n, x):
...     for i in range(n):
...         x = f(x)
...     return x
...
>>> apply_n_times(lambda x: x*x, 3, 2)
256
```

Of course, number of parameters in lambdas is not limited to one; in the example below lambda has two parameters and returns a two-element tuple with the sum of squares of the arguments as the first element and the square of their sum as the second one:

```
>>> myfun = lambda x, y: (x**2+y**2, (x+y)**2)
>>> myfun(4, -1)
(17, 9)
```

Lambdas, being just objects, can be passed or returned to/from functions. In the example below, the **quad** function accepts three coefficients of a quadratic function $ax^2 + bx + c$ which is then returned as a lambda:

```
def quad(a, b, c):
    print('id of c is:', id(c))
    return lambda x: a*x*x + b*x + c

f, g, h = quad(1, 1, 1), quad(2, -1, 3), quad(-1, 4, -2)

print('\n', f(2), g(2), h(2))
print()
print('h.__closure__ is:', h.__closure__, '\n')
print('id of h.__closure__[2].cell_contents is:',
        id(h.__closure__[2].cell_contents))
print('h.__closure__[2].cell_contents is:',
    h.__closure__[2].cell_contents)
```

The program prints

```
id of c is: 140437638578416
id of c is: 140437638578480
id of c is: 140437638578320

 7 9 2

h.__closure__ is:
(<cell at 0x7fba2e1fbcd0: int object at 0x7fba2f8f80b0>,
 <cell at 0x7fba2e1fbca0: int object at 0x7fba2f8f8150>,
 <cell at 0x7fba2e1fbc70: int object at 0x7fba2f8f8090>)

id of h.__closure__[2].cell_contents is: 140437638578320
h.__closure__[2].cell_contents is: -2
```

The **quad** prints also the ID of variable c which is *local* inside this function, so it should cease to exist when the function exits.

However, we then investigate the object representing the **h** lambda. Its attribute __closure__ is a tuple containing three “cells”: they correspond to the three local variables, a, b and c, as they were when the **quad** was executed. Printing the cell_content of the third (with index 2) cell, we can see that its ID and value are exactly the ID and value of c when it was used as local variable during the execution of the function. Therefore, local variables needed in the definition of a function (lambda in particular) are "remembered" in the objects representing the function, even when they seemingly ceased to exist after returning.

A closure is created when a function (lambda) is defined inside a function and "remembers" references to current values of local variables. When it's not in a function, no closure is created.

Let's look at the following snippet of code. Note that we create a lambda which uses a, b and c although they don't exist yet! The lambda contains only their *names:* their values will always correspond to *current* values when the lambda is invoked, so their modifications between calls will be visible, even when they refer to completely new objects of different type:

```
1    quad = lambda x: a*x**2 + b*x + c # a, b, c do not exist!
2    a = 1; b = 1; c = 1               # ints
3    print(quad(-2))                   # prints 3
4    a = 2.5; b = 2.5; c = -0.75       # now floats
5    print(quad(-2))                   # prints 4.25
6    print(quad.__closure__)           # prints None
```

Here, in line 4, we redefine a, b and c, so they refer to different objects of different type (now they are **floats**) and when the lambda **quad** is executed, *new* values are used. No closures are created.

However, if the same code is used in a function, the closure *will* be created

```
def d():
    quad = lambda x: a*x**2 + b*x + c # a, b, c do not exist!
    a = 1; b = 1; c = 1               # ints
    print(quad(-2))                   # prints 3
    a = 2.5; b = 2.5; c = -0.75       # now floats
    print(quad(-2))                   # prints 4.25
    print(quad.__closure__)           # now closure exists!
d()
```

as seen from the output of this program:

```
3
4.25
(<cell at 0x7f5ef349b820: float object at 0x7f5ef3deddb0>,
 <cell at 0x7f5ef349b850: float object at 0x7f5ef3deddb0>,
 <cell at 0x7f5ef349b880: float object at 0x7f5ef3ded3f0>)
```

Lambdas, being functions, can be passed to functions and returned from functions — these functions may be lambdas themselves. Let us consider the following example:

```
1    cmp = lambda f, g: lambda x: f(g(x)) # composition
2
3    fun1 = cmp(lambda x: x-1,  lambda x: x**2)
4    fun2 = cmp(lambda x: x**2, lambda x: x-1)
5
6    print(fun1(3), fun2(3))              # prints 8 4
```

In line 1, we create a lambda which has two parameters, f and g. They both are supposed to be one-argument functions. The lambda's value is also a lambda — in

our case corresponding to the composition $f \circ g$ (recall that the function composition is defined as $(f \circ g)(x) = f(g(x))$). The lambda created is given the name **cmp**.
In lines 3 and 4, we invoke **cmp** passing two functions, as required, and these functions are also defined as lambdas: one corresponds to function $x \mapsto x - 1$ and the second to $x \mapsto x^2$ (we pass them to **fun1** and to fun2 in different order). Now **fun1** represents the function $x \mapsto x^2 - 1$ and fun2 $x \mapsto (x-1)^2$, so the result printed is '8 4'.

Lambdas are often used when calling a function which expects a function as one of its arguments. For example, the **sort** method of lists needs to know according to what the elements are to be sorted: this information is passed as a function, which applied to elements of the list yields values that are to be used for comparing these elements. For example,

```
>>> ls = [('Joe', 182, 79), ('Sue', 173, 59), ('Liz', 175, 62)]
>>> ls.sort(key=lambda person: person[2])
>>> ls
[('Sue', 173, 59), ('Liz', 175, 62), ('Joe', 182, 79)]
```

sorts list of 3-tuples, representing names, heights and weights of persons, according to the third element (index 2: weight):

```
[('Sue', 173, 59), ('Liz', 175, 62), ('Joe', 182, 79)]
```

Although the body of a lambda must be a single expression, it can, in particular, contain invocation of itself — lambdas can be recursive, although for this to be possible, they have to have a name. Recursive lambdas are not very common because they tend to be hard to understand, but they are possible. For example, the factorial function can be defined as:

```
>>> fac = lambda n: 1 if n <= 1 else n*fac(n-1)
>>> fac(5)
120
>>> fac(45)
119622220865480194561963161495657715064383733760000000000
```

On the right-hand side of the assignment on the first line, the name fac is undefined. However, it *will* be known after the assignment, when the lambda is *used.*

5.4 Some "special" functions.

In particular, lambdas are used with **filter**, **map**, **reduce**, **accumulate**[1] and also **sort**/**sorted** functions.
They are used as functions, although, strictly speaking, some of them, as **map** and **filter**, are actually classes (but classes *are* callable, so this it isn't so strange):

```
>>> lst = [1,2,3,4,5,6]
>>>
>>> f = filter(lambda n: n%2 != 0, lst)
>>> type(f)
<class 'filter'>
>>> f
```

[1] These are basic functions in functional languages, like Haskell.

```
<filter object at 0x7fd978400e20>
>>> list(f)
[1, 3, 5]
>>>
>>> m = map(lambda n: n**n, lst)
>>> type(m)
<class 'map'>
>>> m
<map object at 0x7fd978403f10>
>>> list(m)
[1, 4, 27, 256, 3125, 46656]
```

5.4.1 *filter*

The **filter** function takes an iterable (something that can be iterated over) and a function (a predicate) which for any element of the iterable yields **True** (or anything “truthy”) or **False** (or anything “falsy”). It returns an iterable yielding those elements form the input iterable, for which the predicate gives **True**. If the predicate is **None**, it defaults to identity function, which will then leave only “truthy” elements while removing the “falsy” ones.

For example

Listing 21 AAZ-Filter/Filter.py

```python
#!/usr/bin/env python

def is_prime(n):
    if n <= 1: raise ValueError
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    k = 5
    while k**2 <= n:
        if n % k == 0 or n % (k+2) == 0: return False
        k += 6
    return True

lst = [1, 5, 1, 0, '', {}, 2]

print(type(filter))           # <class 'type'>
a = filter(None, lst)
print(a)                      # <filter object at 0x7f569fabbc40>

for e in filter(None, lst):
    print(e, end=' ')         # only 'truthy': 1 5 1 2
print()

lst = ["Kyle", "Sue", "Mary", "Ada"]
for e in filter(lambda s: len(s) > 3, lst):
    print(e, end=' ')         # Kyle Mary
print()
```

```

tup = (3, 9, 12, 11, 27, 83, 787)
for e in filter(lambda n: is_prime(n), tup):
    print(e, end=' ')         # 3 11 83 787
print()

dic = dict(Alice = 15, Jane = 21, Kyle = 12, Mary = 28)
d = dict(filter(lambda item: item[1] >= 18, dic.items()))
print(d)                      # {'Jane': 21, 'Mary': 28}
```

Note that in the last example, we passed the resulting iterable directly to the constructor of a dictionary.

5.4.2 *map*

The **map** function takes a function and one or more input iterable objects and creates an iterable of values obtained by applying the given function to values of input iterable(s). The function must be of the same arity (number of parameters) as is the number of iterables provided. If there are more than one input iterables, the resulting iterable will generate as many tuples of values as there are elements in the shortest of the input iterables.

For example

```
lst = [1, 4, 7, 2, 6]
m = map(lambda x: (x-1)**2, lst)

print(map)         # <class 'map'>
print(m)           # <map object at 0x7f4c7f403a60>
print(list(m))     # [0, 9, 36, 1, 25]
```

Another example, where we apply formatting (see sec. ??, p. ??) to elements of a list

```
lsf = [1.2345, 1.2223e2]
print(list(map(lambda x: f'{x:.1f}', lsf))) # ['1.2', '122.2']
```

or, with more input iterables

```
lsti = [ 1,    3,     2,       2,    2]
lsts = ['a', 'bc', 'def', 'ghij']

m = map(lambda x, y: x*y, lsti, lsts)
print(map)         # <class 'map'>
print(m)           # <map object at 0x7ffaf3583a90>
print(list(m))     # ['a', 'bcbcbc', 'defdef', 'ghijghij']
```

(we get four elements, as the shorter of the two lists has 4 elements).

5.4.3 *reduce*

The **reduce** function (from the ***functools*** module) takes a *binary* function, an iterable and, optionally, an initializer (seed) with the default value **None**.

If the initializer is *not* None or omitted, it becomes the initial value of the so called **accumulator**. Then the function is applied iteratively on the accumulator and consecutive elements of the iterable assigning the result back to the accumulator after each application, thus reducing the iterable to a single value:

```
iterable = [1, 2, 3, 4]
init = 0
fun = lambda x, y: x + y

acc = init
for e in iterable:
    acc = fun(acc, e)
print(acc)                                      # 10

  # now with reduce
from functools import reduce
print(reduce(lambda x, y: x+y, iterable, 0))    # 10
```

With two arguments (i.e., when initializer is None), acc will be initialized with the value of the first element and the first application of the function will be on two first elements:

```
iterable = [1, 2, 3, 4]
init = 0
fun = lambda x, y: x + y

it = iter(iterable)
acc = next(it)                               # first element
for e in it:                                 # it, not iterable
    acc = fun(acc, e)
print(acc)                                   # 10

  # now with reduce
from functools import reduce
print(reduce(lambda x, y: x+y, iterable))    # 10
```

In the two examples above, the results are the same, because in the first of them, we took the the neutral element of addition as the seed; with another value, the results will be, of course, different:

```
from functools import reduce

iterable = [1, 2, 3, 4]

print(reduce(lambda x, y: x+y, iterable))    # 10
print(reduce(lambda x, y: x+y, iterable, 5)) # 15
```

Other examples:

```
from functools import reduce

lsi = [3, 6, -1, 3, 7, 2]
```

```
mn = reduce(lambda x, y: min(x, y), lsi) # min
mx = reduce(lambda x, y: max(x, y), lsi) # max
nb = reduce(lambda x, y: x+1, lsi, 0)    # number of elems

print(mn, mx, nb)          # -1 7 6


lss = ['Kate', 'Joanna', 'Sue', 'Mary']

  # longest word
ln = reduce(lambda x, s: x if len(x) > len(s) else s, lss)
  # initials
ii = reduce(lambda i, s: i + s[0], lss, '')
  # length of the longest word
lo = reduce(lambda l, s: l if l > len(s) else len(s), lss, 0)

print(ln, ii, lo)          # Joanna KJSM 6
```

5.4.4 *accumulate*

The **accumulate** function (from the ***itertools*** module) is similar to **reduce**, but instead of returning a single value, it returns an iterable yielding all "intermediate" results. The first argument of **accumulate** is an iterable, and the second is a binary function. By default it corresponds to just addition (**operator.add**), so these two invocations

```
from itertools import accumulate
lsi = [1, 2, 3]

print(list(accumulate(lsi)))                     # [1, 3, 6]
print(list(accumulate(lsi, lambda x, y: x + y))) # [1, 3, 6]
```

are equivalent.

In the example below, the numbers are payments on a saving account; every month the interest 5% of the accumulated capital is added first and then the amount of the current payment: the returned iterable yields the account balance after each payment (we used **map** to format the numbers):

```
from itertools import accumulate

lsi = [150, 200, 300, 350]

a = accumulate(lsi, lambda summ, e: 1.05*summ + e)
print(type(a))
print(list(map(lambda e: round(e,1), a)))
```

The program prints

```
<class 'itertools.accumulate'>
[150, 357.5, 675.4, 1059.1]
```

One can specify also an initial value by passing the argument initial (this is keyword-only parameter) — then the number of resulting elements will be by one greater than of those in the input iterable:

```
from itertools import accumulate

lsi = [150, 200, 300, 350]

a = accumulate(lsi, lambda summ, e: 1.05*summ + e, initial=100)
print(list(map(lambda e: round(e,1), a)))
```

Now the program prints

```
[100, 255.0, 467.8, 791.1, 1180.7]
```

5.4.5 *sorted*

The **sorted** function takes an iterable and returns its (shallow) copy which is sorted. There is also a *method,* **sort**, of class **list** which when invoked on a list, sorts it. Therefore, the list is rearranged, not copied, and nothing is returned (i.e., **None** is returned).

Important: Algorithm used for sorting is not fixed once for all[1] and can be changed in the future, but it *is* guaranteed to be stable, i.e., elements that are equal, retain their relative order after sorting.

By default, **sorting** uses the < operator to compare elements, so it will work for numbers (but not **complex**) and strings, as well as other types for which this operator is defined (for example, for which **__lt__** method is defined):

```
lsiorig = [7, 2, 8, 1, 5]
lsorted = sorted(lsiorig)
print('original', lsiorig) # [7, 2, 8, 1, 5]
print('  sorted', lsorted) # [1, 2, 5, 7, 8]
```

Besides a sequence to be sorted, there are two additional parameters of **sorted**, both are optional and keyword-only.

One of them is reverse, with obvious meaning and default value **False**:

```
lsiorig = [7, 2, 8, 1, 5]
lsorted = sorted(lsiorig, reverse=True)
print('original', lsiorig) # [7, 2, 8, 1, 5]
print('  sorted', lsorted) # [8, 7, 5, 2, 1]
```

The third (also keyword-only) parameter of **sorted** is called key. The corresponding argument should be a unary function[2] which will be applied to elements and then the results will be compared instead of original values. Note however that these results will be used only for comparing, elements of the input sequence themselves will not be modified, only rearranged.

In the example below , we sort tuples in two ways: by their first element (name) and then by the second (age):

[1] For many years it was Timsort – a hybrid algorithm based on merge sort and insertion sort (invented by Tim Peters in 2002). This algorithm is also used in Java and in Android system. In Python 3.11, it was replaced by the new algorithm – Powersort (I. Munro i S. Wild).

[2] Sometimes called *projector*.

```
ltup = [('Jane', 29), ('Alice', 31), ('Betty', 7)]
print(sorted(ltup, key=lambda t: t[0]))
print(sorted(ltup, key=lambda t: t[1]))
```

prints

```
[('Alice', 31), ('Betty', 7), ('Jane', 29)]
[('Betty', 7), ('Jane', 29), ('Alice', 31)]
```

In the following example, we first sort without specifying any key function, so the < will be used for strings (recall that all capital letters are “smaller” than all lowercase letters). Then we sort, but this time passing the lower method as the key function: elements of the list will not be modified, but comparisons will now be case-insensitive. Finally, we print the original list just to show that it remained unmodified — sorted returns the sorted *copy:*

```
lstring = ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
lsorted = sorted(lstring)
ltolowe = sorted(lstring, key = lambda s: s.lower())
print('   original', lstring)
print(' sort as is', lsorted)
print('case insens', ltolowe)
print('   original', lstring)
```

prints

```
   original ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
 sort as is ['Hugh', 'Jane', 'alice', 'bertha', 'henry']
case insens ['alice', 'bertha', 'henry', 'Hugh', 'Jane']
   original ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
```

Very often criteria for comparison are more complex; for example we want to compare using one, “primary”, criterion, but then for elements which come out to be equal, use other, “secondary”, criteria in addition. To solve this problem, we can leverage the fact, that tuples *are* comparable (using < operator) if only their consecutive elements are. Therefore, the key function can return tuples, which will be then compared lexicographically.

For example here, we sort strings by their lengths. However, those with equal lengths will be additionally sorted alphabetically and case-insensitively:

```
ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']
print(sorted(ls, key=lambda s: (len(s), s.lower())))
```

prints

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

Alternatively, thanks to the guaranteed stability of sorting algorithm, we can first sort by the *secondary* criteria and then sort the result using the *primary* ones.

For example, in the snippet below, we sort strings by their lengths. However, those with equal lengths will be additionally sorted alphabetically and case-insensitively:

```
ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']
    # secondary: alphabetically, case-insensitively
ls = sorted(ls, key=lambda s: s.lower())
    # and now primary: length
ls = sorted(ls, key=lambda s: len(s))
print(ls)
```

prints as before

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

In many languages, sorting is based on binary comparing functions which, given two objects, return anything negative if the first is to be considered “smaller”, anything positive if it’s the second one, and 0 if they are deemed equal. For those used to this kind of comparisons, there is a special function, **cmp_to_key** from the ***functools*** module, which takes a binary comparing function, as described above, and returns a callable that given *one* argument yields a value which can be used as the sort key:

```
from functools import cmp_to_key

def my_cmp(s1, s2):
      # by length
    if (d := len(s1) - len(s2)) != 0: return d
      # alphabetically, case-insensitively
    if   s1.lower() < s2.lower(): return -1
    elif s2.lower() < s1.lower(): return +1
    else                        : return  0

ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']

ls = sorted(ls, key=cmp_to_key(my_cmp))
print(ls)
```

prints again

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

5.5 Function decorators.

A decorator is a function (more generally – a callable) that takes another callable[1] as its argument and returns also a callable.
For example:

```
def decorator(func):
    return func

def adder(a, b):
    """Return sum of arguments."""
    return a + b
```

[1] Classes and their methods are callable too, so they also can be decorated. In the examples below, we will use just functions, though.

```
add = decorator(adder)
print(add(5, 7))        # 12
print(add.__name__)     # adder
```

In this trivial example, **add** behaves just as **adder**, although its __name__ property is adder, as the original name of the function wrapped by the **decorator**. However, we can make it less trivial by returning a function which is like the original, but with some features added or modified, and this is so useful that it gained a special syntax which makes it easier and more readable: instead of invoking the decorator function explicitly, we *annotate* our function with the name of the decorator:

```
def decorator(func):
    return func

@decorator
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(5, 7))  # 12
```

One can say that

```
@decorator
def adder(a, b):
    """Return sum of arguments."""
    return a + b
```

is equivalent to

```
def adder(a, b):
    """Return sum of arguments."""
    return a + b
adder = decorator(adder)
```

Normally, of course, the returned function is not just identical to the input function but has some features added or modified. These could be, among others, registering a function somewhere, modifying its input or output, logging or counting its invocations, caching the results of its invocations, etc.

Consider

```
def counting(func):                         # decorator
    counter = 0
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    return modified_func

@counting
def adder(a, b):
    """Return sum of arguments."""
    return a + b
```

```
print(adder(-1, 2), adder(5, 7))           # 1 12
print(adder.__closure__[0].cell_contents) # 2
print(adder.__name__)                      # modified_func
print(adder.__doc__)                       # None
```

Now the decorator **counting** returns the original function but with a counter of invocations added to its closure (see p. 76). However, there is a little problem: its __name__ property is now modified_func instead of adder and the docstring has been lost. We can fix it “by hand”:

```
def counting(func):                        # decorator
    counter = 0
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    modified_func.__name__ = func.__name__
    modified_func.__doc__  = func.__doc__
    return modified_func

@counting
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(-1, 2), adder(5, 7))           # 1 12
print(adder.__closure__[0].cell_contents) # 2
print(adder.__name__)                      # adder
print(adder.__doc__)                       # Return sum of arguments.
```

However, this would be cumbersome and we wouldn’t be sure if all function metadata has been correctly copied. The solution is to use special function **wrap** from the ***functools*** module (which is itself a decorator)

```
import functools

def counting(func):                        # decorator
    counter = 0
    @functools.wraps(func)                 # wrapping tool
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    return modified_func

@counting
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(-1, 2), adder(5, 7))           # 1 12
print(adder.__closure__[0].cell_contents) # 2
print(adder.__name__)                     # adder
print(adder.__doc__)                      # Return sum of arguments.
```

Adding data to the metadata of a function makes it possible to implement the so called *memoization*, i.e., ability of remembering information from previous invocations of a function.[1]

Let us consider the Fibonacci numbers; as we know, calculating them recursively is extremely ineffective, as the number of invocations grows as 2^N for the N-th Fibonacci number. But it becomes trivial and practically immediate, if we can remember all previous results for smaller numbers.

In the example below, the decorator **memo** adds a dictionary cache as attribute of the decorated function. The cache remembers results with tuples of arguments as keys, so it can immediately return the answer if the result with given arguments has already been calculated before. We can access this cache directly as __wrapped__.cache attribute of the decorated function:

Listing 22 ABS-Deco/Deco.py

```python
#!/usr/bin/env python

import functools

def memo(func):
    func.cache = {}
    @functools.wraps(func)
    def memo(*args):
        if args not in func.cache:
            func.cache[args] = func(*args)
        return func.cache[args]
    return memo

@memo
def fibo(n):
    if n < 2: return n
    else    : return fibo(n - 1) + fibo(n - 2)

for i in range(50, 201, 50):
    print(f'fibonacci[{i:3}] = {fibo(i)}')

print(fibo.__wrapped__.cache)
```

The program prints

```
fibonacci[ 50] = 12586269025
fibonacci[100] = 354224848179261915075
fibonacci[150] = 9969216677189303386214405760200
```

[1] As a matter of fact, such decorator already is implemented in the standard library as lru_cache (**l**east **r**ecently **u**sed cache) from the ***functools*** module — see documentation.

```
fibonacci[200] = 280571172992510140037611932413038677189525
{(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8,
 ...
 (197,): 66233869353085486281758142155705206899077,
 (198,): 107168651819712326877926895128666735145224,
 (199,): 173402521172797813159685037284371942044301,
 (200,): 280571172992510140037611932413038677189525}
```

Note that the **memo** decorator can be applied to very wide spectrum of functions, not necessarily to binary and/or recursive.[1]

Decorators can be nested: the outer decorator is then applied to the result of the inner decorator, as in the example below:

Listing 23 ABT-DecoNest/DecoNest.py

```python
#!/usr/bin/env python

import functools

def italic(func):
    @functools.wraps(func)
    def _f(s):
        return '<i>' + func(s) + '</i>'
    return _f

def strong(func):
    @functools.wraps(func)
    def _f(s):
        return '<strong>' + func(s) + '</strong>'
    return _f

@italic
def ital(s):
    return s

@strong
def stro(s):
    return s

@italic
@strong
def itst(s):
    return s

@strong
@italic
def stit(s):
    return s

```

[1] Do not try to run the above program without caching!

```python
print(ital(" just ital "))
print(stro(" just strong "))
print(itst(" ital and strong "))
print(stit(" strong and ital "))
```

which prints

```
<i> just ital </i>
<strong> just strong </strong>
<i><strong> ital and strong </strong></i>
<strong><i> strong and ital </i></strong>
```

Very often it would be convenient to pass additional information to a decorator, so its functionality depends somehow on this information. This is possible by constructing a “decorator factory” — a function which depends on some arguments and returns an actual decorator with which we decorate a function.

In the example below, **rangecheck** function[1] accepts two arguments, mn and mx, which define the range $[mn, mx]$ to which all arguments of a function are supposed to belong. For given mn and mx, the **rangecheck** function creates and returns the **checker** function: this is the decorator proper, which, being a decorator, takes a function and returns a function (here called **wrap**) that will “replace” a decorated function (**color** on line 18, and **fRGBColor** on line 22). What is important, references mn and mx can be used inside **wrap** (according to LEGB rule) and hence the resulting decorated function will depend on them:

Listing 24 ABU-DecoParam/DecoParam.py

```python
 1  #!/usr/bin/env python
 2
 3  import functools
 4
 5  def rangecheck(mn, mx):
 6      def checker(func):                  # decorator to be returned
 7          @functools.wraps(func)
 8          def wrap(*args):                # wraper of func
 9              for i in args:
10                  if i < mn or i > mx:
11                      raise ValueError(
12                          f'Argument {i} in {args} is out of range')
13              return func(*args)          # if args ok, just call func
14          return wrap
15      return checker
16
17  @rangecheck(0, 255)
18  def color(r, g, b):
19      return f'Color {r=}, {g=}, {b=}'
20
21  @rangecheck(0,1)
```

[1] This function is *not* a decorator: it *returns* a decorator.

```python
def fRGBColor(r, g, b):
    return f'fColor {r=:.1f}, {g=:.1f}, {b=:.1f}'

try:
    print(color(  2, 255,   0))
    print(color(  0, 128, 34))
    print(color(103,  12, 88))
    print(color(  0, 256,   0))
except ValueError as e:
    print(e)

print()

try:
    print(fRGBColor(0.0, 0.9, 0.0))
    print(fRGBColor(0.5, 0.8, 0.0))
    print(fRGBColor(0.2, 0.1, 0.8))
    print(fRGBColor(1.1, 0.9, 0.3))
except ValueError as e:
    print(e)
```

The program prints

```
Color r=2, g=255, b=0
Color r=0, g=128, b=34
Color r=103, g=12, b=88
Argument 256 in (0, 256, 0) is out of range

fColor r=0.0, g=0.9, b=0.0
fColor r=0.5, g=0.8, b=0.0
fColor r=0.2, g=0.1, b=0.8
Argument 1.1 in (1.1, 0.9, 0.3) is out of range
```

Decorators are executed when the module is loaded (imported), so they are “ready to use” inside the module. The example below illustrates this feature:

Listing 25 ABY-DecoTime/DecoTime.py

```python
#!/usr/bin/env python

import functools

def decorator(func):
    print('Decorating', func.__name__)
    @functools.wraps(func)
    def wrap(arg):
        val = func(arg)
        print('Decorated', func.__name__, ':', arg, '->', val)
        return val
    return wrap
```

```python

@decorator
def funA(x):
    return x

@decorator
def funB(x):
    return x*x

print('\n*** Starting the program\n')

print('Calling funA with arg =', 4)
v = funA(4)

print('Calling funB with arg =', 5)
v = funB(5)
```

As we can see from the printout

```
Decorating funA
Decorating funB

*** Starting the program

Calling funA with arg = 4
Decorated funA : 4 -> 4
Calling funB with arg = 5
Decorated funB : 5 -> 25
```

the lines

```
Decorating funA
Decorating funB
```

appeared *before* entering the program proper.

Chapter 6

Comprehensions and generators

Sequences are the basic data structures in Python, and so are iterables and iterators. As we remember, anything that we can pass to the built-in **iter** function and get an iterator *is* iterable. Iterator in turn is something that we can pass to the built-in **next** function and get the next item, or, if there are no more items to return, get the **StopIteration** exception. For the **next** function to work, iterators should provide the **__next__** *method;* if they don't also **__getitem__** (taking a zero based index) will be tried.

Iterators are themselves iterable, i.e., passed to **iter** function, they return iterator, most often just themselves.

Normally, we don't have to call **iter** and **next** "manually". Instead, we use **for**-loop which is really a **while** loop, but does what's needed under the hood:

```
iterable = [1, 2, 3] # lists are iterable

print('for loop:   ', end='')

for e in iterable:
    print(e, end=' ')

## equivalent while loop

print('\nwhile loop: ', end='')

itr = iter(iterable)
while True:
    try:
        e = next(itr)
    except StopIteration:
        break
    else:
        print(e, end=' ')
del itr
```

prints

```
for loop:   1 2 3
while loop: 1 2 3
```

Comprehensions and generators, as loops, are used for performing iteration. They both take items one by one from a source and do something with each of them in turn.

6.1 Comprehensions

Comprehensions[1] in Python provide a concise syntax for creating lists, sets and dictionaries from other iterables in a declarative/functional style.
They always act on existing iterables and iterating over their elements produces results that are then collected to a list, a set, or a dictionary. As loops, they can be nested.

[1] The term borrowed from *set comprehension* in mathematics.

It is also possible to define some condition that must be met for the result of a given iteration to contribute to the final collection.

6.1.1 List comprehension

List comprehensions have the form

```
[ expr for var in iterable]
```

where iterable is an iterable (e.g., a list), var is the name of a variable that will hold references to consecutive elements of the iterable, and expr is an expression (usually depending on var) the value of which will become an element of the resulting list.
For example

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> [val**2 for val in iterable]
[1, 4, 9, 16, 25, 36]
```

or

```
>>> iterable = [ [1, 2], [3], [4, 5, 6] ]
>>> [sum(v) for v in iterable]
[3, 3, 15]
```

After an iterable, one can add a condition which will decide if expr from this iteration will or will not be included into the resulting list:

```
>>> [v**2 for v in range(1,10) if v % 2]
[1, 9, 25, 49, 81]
```

Here, we take squares of element of the range (numbers from the interval $[1, 9]$) but only of those which are odd (so v%2 is 1, which is interpreted as **True**).
Note that the variable used for iteration (in our case v) is local inside the comprehension; it doesn't conflict with possibly existing another variable of the same name in the surrounding scope:

```
>>> v = 'Hello, World'
>>> [v**2 for v in range(1,10) if v % 2]
[1, 9, 25, 49, 81]
>>> v
'Hello, World'
```

Iterations can be nested, and for each of them we can add a condition:

```
>>> ls = [ [1, 2], 'abc', [3, 4, 5, 6,], 'xy' ]
>>> [k for v in ls if isinstance(v, list) for k in v if k%2 == 0]
[2, 4, 6]
```

which is equivalent to

```
ls = [ [1, 2], 'abc', [3, 4, 5, 6,], 'xy' ]
res = []
for v in ls:
    if isinstance(v, list):
        for k in v:
            if k % 2 == 0:
                res.append(k)
print(res)
```

and prints

```
[2, 4, 6]
```

In order to get used to comprehensions, let us consider examples.

Example 1
Squares of numbers:

```
>>> [x*x for x in range(2,11)]
[4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 2
Squares of even numbers:

```
>>> [x*x for x in range(2, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

Example 3
Different powers of even and odd numbers:

```
>>> [x*x if x%2 else x*x*x for x in range(1,7)]
[1, 8, 9, 64, 25, 216]
```

Note the expression x%2 as a condition — it will be interpreted as True for odd numbers.

Example 4
Converting strings to numbers:

```
>>> [float(x) for x in ['3.3' , '1.25e2', 'NaN', '-1', '-INF']]
[3.3, 125.0, nan, -1.0, -inf]
```

Example 5
Selecting only numbers or only strings:

```
>>> [x for x in [1, 'a', 'bc', 1.2e2] if isinstance(x, int | float)]
[1, 120.0]
>>> [x for x in [1, 'a', 'bc', 1.2e2] if isinstance(x, str)]
['a', 'bc']
```

Example 6
Number of digits in huge numbers:

```
>>> from math import factorial
>>> [len(str(factorial(x))) for x in range(100, 111)]
[158, 160, 162, 164, 167, 169, 171, 173, 175, 177, 179]
```

Example 7
Prime numbers <= 31

```
>>> [k for k in range(2, 32)
...     if all(k%m != 0 for m in range(2, int(k**0.5)+1))]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Note: the **all** takes a predicate and an iterable and yields **True** if, and only if all elements meet the predicate.

Example 8
Removing spaces:

```
>>> ''.join([x for x in 'to be or not to be' if x != ' '])
'tobeornottobe'
```

Example 9
Common elements of two list:

```
>>> ls1 = [1, 'a', 3, 'b', 5, 'c']
>>> ls2 = [4, 'c', 5, 'f', 8]
>>> [a for a in ls1 if a in ls2]
[5, 'c']
```

Example 10
Max elements of sequences:

```
>>> lst = [[1, 4, 3, 2], [19 , -1, 2], [-1, -20, -6], [5, 9]]
>>> [max(x) for x in lst]
[4, 19, -1, 9]
```

Example 11
Converting a matrix of strings to numbers:

```
>>> ls = [['4', '8'],
...       ['4', '2', '28'],
...       ['1', '12'],
...       ['3', '6', '2']]
>>> [[int(e) for e in row] for row in ls]
[[4, 8], [4, 2, 28], [1, 12], [3, 6, 2]]
```

Note that here the expression is itself a comprehension list.

Example 12
Flattening lists:

```
>>> lst = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
>>> [ch for ls in lst for ch in ls]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Example 13
Couples with names starting with the same letter:

```
>>> boys  =  ['Adam', 'Burt', 'Joe', 'Jim', 'Sam']
>>> girls =  ['Bea', 'Jill', 'Ava', 'Eve']
>>> [(she, he) for she in girls for he in boys if she[0] == he[0]]
[('Bea', 'Burt'), ('Jill', 'Joe'), ('Jill', 'Jim'), ('Ava', 'Adam')]
```

Example 14
Pythagorean triplets with a < b < c <= 50:

```
>>> from math import gcd
>>> def gcd3(a, b, c): return gcd(a, gcd(b, c))
...
>>> [(a, b, c) for a in range(1,49)
...            for b in range(a+1, 50)
...            for c in range(b+1, 51)
...                if a*a + b*b == c*c and gcd3(a, b, c) == 1]
[(3, 4, 5), (5, 12, 13), (7, 24, 25), (8, 15, 17),
(9, 40, 41), (12, 35, 37), (20, 21, 29)]
```

Note that we select only triangles pair-wise dissimilar.

6.1.2 Set comprehension

Set comprehensions are similar to list comprehensions but values of expr are collected to sets (so without duplicates) rather than to lists. Here, we use curly braces instead of square brackets.

Examples:

Example 1
Unique words:

```
>>> ss = ['Warsaw Paris Berlin', 'Praha Warsaw Berlin Caen']
>>> {word for s in ss for word in s.split()}
{'Paris', 'Warsaw', 'Praha', 'Berlin', 'Caen'}
```

Example 2
Common elements of lists:

```
>>> list1 = [1, 3, 7, 4]
>>> list2 = [3, 4, 5, 7]
>>> list3 = [4, 5, 6, 7]
>>> {x for x in list1 if x in list2 and x in list3}
{4, 7}
```

Example 3
Set of Polish vowels in a text:

```
>>> string = 'Pozbądź się zbędnych kilogramów'
>>> {x for c in string if (x := c.lower()) in 'aąeęiouy'}
{'ę', 'y', 'ą', 'a', 'i', 'o'}
```

Example 4
Unique letters in a text:

```
>>> string = 'carrot: 20; apples: 30'
>>> {c for c in string if c.isalpha()}
{'s', 'r', 'e', 'c', 'p', 'a', 'l', 't', 'o'}
```

Example 5
Cartesian product of sets:

```
>>> l1 = [1, 2, 3]
>>> l2 = ['a', 'b']
>>> {(x, y) for x in l1 for y in l2}
{(1, 'b'), (3, 'a'), (2, 'a'), (2, 'b'), (1, 'a'), (3, 'b')}
```

Example 6
Set of numbers from a list:

```
>>> vals = [1, 'a', 2, 'b', 3, 'c']
>>> {x for x in vals if isinstance(x, int | float)}
{1, 2, 3}
```

6.1.3 Dictionary comprehension

Finally, colon separated *pairs* of values of two expressions

```
{ expr1 : expr for var in iterable }
```

will be collected to a dictionary (as for sets, comprehension expression must be enclosed in curly braces).

A few examples:

Example 1
Dictionary number → parity:

```
>>> nums = [1, 2, 3, 4]
>>> {n : 'odd' if n%2 else 'even' for n in nums}
{1: 'odd', 2: 'even', 3: 'odd', 4: 'even'}
```

Example 2
Letter counts in a text:

```
>>> string = 'to be or not to be'
>>> {c : string.count(c) for c in set(string) if c.isalpha()}
{'r': 1, 'n': 1, 'b': 2, 'e': 2, 't': 3, 'o': 4}
```

Example 3
Dictionary name → its length:

```
>>> names = ['Argentina', 'Poland', 'Spain', 'Italy']
>>> {name: len(name) for name in names}
{'Argentina': 9, 'Poland': 6, 'Spain': 5, 'Italy': 5}
```

Example 4
Tuple to dictionary:

```
>>> lst = [('a', 'Alice'), ('b', 'Bea'), ('c', 'Cecilia')]
>>> {k: v for k, v in lst}
{'a': 'Alice', 'b': 'Bea', 'c': 'Cecilia'}
```

Example 5
Marks from scores:

```
>>> scores = {'Eve': 83, 'Bob': 47, 'Sue': 65, 'Dave': 77}
>>> {name:
...     '5'  if score >= 90 else
...     '4+' if score >= 80 else
...     '4'  if score >= 70 else
...     '3+' if score >= 60 else
...     '3'  if score >= 50 else '2'
...  for name, score in scores.items()}
{'Eve': '4+', 'Bob': '2', 'Sue': '3+', 'Dave': '4'}
```

Example 6
Filtering a dictionary:

```
>>> dct = {'a': 9, 'b': 12, 'c': 7, 'd': 14}
>>> {k: v for k, v in dct.items() if v <= 10}
{'a': 9, 'c': 7}
```

Example 7
Reversing dictionary:

```
>>> dct = {'Mary': 'John', 'Kate': 'Bill', 'Sophia': 'Kevin'}
>>> {v : k for k, v in dct.items()}
{'John': 'Mary', 'Bill': 'Kate', 'Kevin': 'Sophia'}
```

6.2 Generator functions

Generator is a special kind of objects that can be used to lazily produce a sequence of values. Lazily means that consecutive elements are created and returned only when they are needed, not all of them in advance. Generators are *iterable* — they provide the **__iter__** function which returns an iterator. Main benefits of generators are:

- They pause execution until the next value is demanded; when paused, they do nothing without consuming processor time and therefore are completely “lazy”.
- They process only values that have been demanded; values that are not needed will never be even evaluated.
- They can represent streams of data of unlimited size, because they do not store values to be yielded anywhere; when asked for, they process just a single value.

On the other hand

- The user doesn’t know the number of values in the remaining stream of data until the generator is exhausted.
- Values yielded by a generator cannot be obtained by indexing or slicing.
- To get a specific item, all preceding values must be fetched.
- You cannot restart or “rewind” a generator

Definition of a generator function looks like a definition of a function. However, if it is a generator, instead of **return**, we use **yield** — by the presence of **yield** statement(s), the interpreter recognizes that this is a generator, not a “plain” function (although **return** may still appear in generator functions). Calling such a “function” behaves differently than calling a “normal” function: invocation returns an object of type **generator** without executing any code!

The returned generator is iterable: it has the __iter__ method which returns an iterator which in turn has __next__ method returning consecutive values “returned” by **yield**. Execution of the code starts when the __next__ method is called for the first time. Every time **yield** is encountered, the value given after this keyword is returned, but the state of the function is “saved” and will be resumed after the next invocation of **next**:

```
>>> def squares():
...     i = 1
...     while True:
...         yield i*i
...         i += 1
...
>>> gen = squares()
>>> type(squares), type(gen)
(<class 'function'>, <class 'generator'>)
>>> hasattr(gen, '__iter__')
True
>>> it = iter(gen)
>>> type(it)
<class 'generator'>
>>> hasattr(it, '__next__')
True
>>> it.__next__()
1
>>> next(it)
4
```

Note how **yield** behaves as **return**, but the execution of the body of the generator is only suspended, and the next invocation of **next** resumes it where it was stopped (until next **yield**). When the flow of control leaves the body of a generator “naturally” or by executing **return**), the **StopIteration** exception is raised:

```
>>> def squares(n):
...     i = 1
...     while i < n:
...         yield i*i
...         i += 1
...
>>> gen = squares(3)
>>> it = iter(gen)
>>> next(it)
1
>>> next(it)
4
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

To stop the iteration, one can also use a “normal” **return** statement — then we will also get **StopIteration** exception:

```
>>> def squares():
...     i = 1
...     while True:
...         yield i*i
...         i += 1
...         if i == 3: return
...
>>> gen = squares()
>>> next(gen)
1
>>> next(gen)
4
>>> next(gen)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Another example — Fibonacci numbers generator:

```
def Fibo(n):
    a, b, count = 0, 1, 0
    while True:
        if (count > n): return
        yield a
        a, b = b, a + b
        count += 1

fib = Fibo(13)
for e in fib:
    print(e, end=' ')
print()
```

prints

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

(note that the **StopIteration** exception has been handled by the **for** loop so we will not see it).

However, there is a difference between using **return** and just exiting the generator. In the former case, we can return *a value*, and it will be remembered as an attribute of the exception object; for example

```
def Fibo():
    a, b, count = 0, 1, 0
    while True:
        if (count > 2): return f'{count=}'
        yield a
        a, b = b, a + b
        count += 1

fib = Fibo()
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
```

prints

```
0
1
1
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
  exec(code, self.locals)
File "/home/werner/p.py", line 13, in <module>
  print(next(fib))
StopIteration: count=3
```

6.2.1 Generator comprehension

Generators can also be created by using the syntax similar to list comprehension; the difference is that now round parentheses are used instead of brackets (this is called **generator expression**):

```
>>> g = (x**2 for x in [2, 4, 7])
>>> type(g)
<class 'generator'>
>>> for i in g:
...     print(i)
4
16
49
```

Note that there is no explicit **yield** statements here and the resulting generator can be iterated over more or less like a list (because it does have methods **__iter__** and **__next__**). But the main advantage of generator is that when a list is created, it is created once with all its elements: generators produce consecutive elements on demand when they are explicitly asked for. Therefore, generators consume virtually no memory and are created very fast.

```
>>> from sys import getsizeof
>>> lst = [x**3 for x in range(10_000_000)]
>>> gen = (x**3 for x in range(10_000_000))
>>> lst.__sizeof__()
89095144
>>> gen.__sizeof__()
88
>>> getsizeof(lst)
89095160
>>> getsizeof(gen)
104
```

As we can see, the **getsizeof** reports slightly larger values than **__sizeof__** method. Actually, it uses the latter internally, but adds some overhead for the garbage collector stuff (experience shows that this is 16 bytes, independently of the object's size).

The example below shows how to create a generator which will generate consecutive Fibonacci numbers:

Listing 26 ABA-GenFib/GenFib.py

```python
#!/usr/bin/env python3

def fibonacci(n):
    curr = 0
    prev = 1
    count = 0
    while count <= n:
        yield curr
        prev, curr = curr, prev + curr
        count += 1

fibogen = fibonacci(100)
print(type(fibonacci), type(fibogen))
for i, f in enumerate(fibogen):
    print('{:3d}:{:22d}'.format(i, f))
```

The program prints

```
    <class 'function'> <class 'generator'>
      0:                     0
      1:                     1
      2:                     1
      3:                     2
      4:                     3
      5:                     5
      6:                     8
      7:                    13
               ...

     95:  31940434634990099905
     96:  51680708854858323072
     97:  83621143489848422977
     98: 135301852344706746049
     99: 218922995834555169026
    100: 354224848179261915075
```

Another example: a generator which reads a text file and yields words read from it but all different, ignoring duplicates:

Listing 27 ABL-UniqueWords/UniqueWords.py

```python
#!/usr/bin/env python

def uniqueLines(inpFile):
    prevWords = set()
    with open(inpFile) as file:
        for line in file:
              # get rid of everything but letters and spaces
            line = ''.join([x.lower() for x in line
                            if x.isalpha() or x == ' '])
            for word in line.split():
                if word not in prevWords:
                    prevWords.add(word)
                    yield word          # yielding new word

for w in uniqueLines('UniqueWords.txt'): print(w)
```

If the ***UniqueWords.txt*** file contains

```
Kraków Paris Zürich
Warszawa Berlin
Paris
Kraków Zürich Warszawa
```

the result will be

```
kraków
paris
zürich
warszawa
berlin
```

And another example: this time we create a generating function **limiter** which takes, as the first argument, another generating function. Then it calls this function, and iterates over the obtained generator only a prescribed number of times (10 in the example).[1] In this way it will be possible to pass to it potentially infinite generators, for example, as below, a generator **Fibo** generating Fibonacci numbers *ad infinitum*, or **Die** which constantly rolls a die:

Listing 28 ABR-GenOfGen/GenOfGen.py

```python
#!/usr/bin/env python

from random import randint

def limiter(generator, n):
    g = generator()
    for i in range(n):
        yield next(g)
```

[1] Somewhat like **Stream::limit** in Java.

```

def Fibo():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b

def Die():
    while True:
        yield randint(1, 6)

for e in limiter(Fibo, 10):
    print(e, end=' ')
print()

for e in limiter(Die, 10):
    print(e, end=' ')
print()
```

The program prints

```
0 1 1 2 3 5 8 13 21 34
2 1 2 5 1 4 3 3 1 2
```

6.3 Communicating with generators

Generators can not only return values but also receive them. We send values (objects) to a generator by invoking the **send** method on it. This method sends a value *to* a generator and also returns a value yielded *from* this generator. In order to receive a value sent by the **send** function, execution of the generator function must be stopped on a line containing the **yield** *expression* — expression, so, for example, on the right-hand side of an assignment. Such generators are sometimes called **coroutines**.

To achieve this situation, we have to "prime" the generator by invoking **next** on it (or sending **None** by its **send** method).

Suppose, there is a **yield** expression of the form `yield expr` on the line where the generator is stopped. Then, after the invocation of **send(v)**:

- The value of v becomes the result of the current `yield expr` expression (but *not* of the expr itself!);[1]
- The execution continues, until the `yield expr` expression is encountered again (or the **StopIteration** exception is raised, if the generator exits without yielding another value);
- Current value of expr is returned to the caller;
- Execution of the generator is again suspended.

Let us consider an example which will show all this in action:

[1] The value of the expression `expr` and the value of the expression `yield expr` are two different things!

Listing 29 ABO-TwoWayGener/TwoWayGener.py

```python
#!/usr/bin/env python

def twoWayGener():
    print("generator started")
    value = 'Init'
    while True:
        print('Ready to recive a value, value =', value)
        x = 10 + (yield value)
        print('Resuming execution; value =', value, 'x =', x)
        value = x

gen = twoWayGener()
ret = next(gen) # (or gen.send(None)) - priming
print('After priming: result =', ret, end='\n\n')

for n in range(3):
    print('Main: sending', n)
    ret = gen.send(n)
    print('Main: result =', ret, end='\n\n')
```

prints

```
generator started
Ready to recive a value, value = Init
After priming: result = Init

Main: sending 0
Resuming execution; value = Init x = 10
Ready to recive a value, value = 10
Main: result = 10

Main: sending 1
Resuming execution; value = 10 x = 11
Ready to recive a value, value = 11
Main: result = 11

Main: sending 2
Resuming execution; value = 11 x = 12
Ready to recive a value, value = 12
Main: result = 12
```

The next example will be a little more advanced. We build a generator which generates random elements from a list passed as the argument. Moreover, by sending non-**None** values to the generator, we can append new elements to the list:

Listing 30 ABP-Chooser/Chooser.py

```python
#!/usr/bin/env python

```

```
from random import choice

def chooser(aList):
    item = None
    while True:
        if item != None:
            if item not in aList:
                aList.append(item)                ❶
            item = yield item                     ❷
        else:
            item = yield choice(aList)            ❸

ch = chooser(['Joe', 'Ben', 'Alice'])
# priming
next(ch)                                          ❹

for i in range(3):
    print(next(ch)) # equivalently: print(ch.send(None))

print('*** Adding Eve and Kenny to the list')
ch.send('Eve')                                    ❺
ch.send('Kenny')                                  ❻

for i in range(4):                                ❼
    print(next(ch)) # equivalently: print(ch.send(None))
```

prints, for example (specific values are random):

```
Alice
Alice
Ben
*** Adding Eve and Kenny to the list
Kenny
Joe
Eve
Alice
```

Let us analyse what's happening here. At first item is None, so, after priming (line ❹), the generator goes to the line ❸ and stops.
Then, in a loop, we send None (**next** is equivalent to sending None) to the generator: the value of the expression `yield choice(aList)` is therefore None, generator resumes, assigns this None to item and goes again to the line ❸, yields the value of `choice(aList)` to the caller, and is suspended again. And this is repeated three times.

Now, we send `'Eve'` (line ❺).
This value becomes the value of the expression `yield choice(aList)` which is assigned to item making it not-None. Execution resumes, goes this time to line ❶, appends `'Eve'` to the list, proceeds to line ❷, returns this `'Eve'` to the caller (who ignores it) and is suspended. Now we send `'Kenny'` (❻). This becomes the value of the expression `yield item` and is assigned to item, so we again reach the lines ❶ and ❷. The next value sent to the generator will be None (from the loop in line ❼), so item will become

None again and we will proceed to line ❸and a new random element of the list will be returned. And this will be repeated 4 times.

Having to "prime" generators before sending data to them may be cumbersome and it's easy to forget about it. The standard method to handle this problem is by using a decorator which takes a coroutine and returns it already primed, as illustrated below. To clarify the logic, the program prints some additional information.

The **longlines** coroutine accepts texts sent to it and prints only those which contain more than n words. In order to use it without priming it first, we decorate it with the **coroutine** decorator: it just calls the generating function to get the generator, calls **next** on it once, and returns the generator that doesn't need priming any more:

Listing 31 ABV-Priming/Priming.py

```python
#!/usr/bin/env python

import functools

def coroutine(func):
    print('** 2 ** Executing coroutine')
    @functools.wraps(func)
    def primed(*args, **kwargs):
        p = func(*args, **kwargs)     # p is a generator
        print('** 4 ** From primed')
        next(p)     # priming
        return p    # returning already primed generator
    return primed

print('** 1 ** Defining longlines')

@coroutine
def longlines(n):
    """Select lines with more than n words."""
    print('** 5 ** Longlines advances to yield')
    while True:
        line = yield
        if len(line.split()) > n: print('Got long line:', line)

print('** 3 ** Main starting, creating generator')
gen = longlines(3)
print(type(gen))
print(type(longlines), longlines.__name__, longlines.__doc__, '\n')

gen.send("one two")
gen.send("uks kaks kolm neli")
gen.send("eins zwei drei")
gen.send("jeden dwa")
gen.send("un deux trois")
gen.send("uno dos")
gen.send("uno due tre quattro")

gen.close()
print('\n** 6 ** Generator closed\n')

gen.send("ichi ni san")
```

The program prints

```
** 1 ** Defining longlines
** 2 ** Executing coroutine
** 3 ** Main starting, creating generator
** 4 ** From primed
** 5 ** Longlines advances to yield
<class 'generator'>
<class 'function'> longlines Select lines with more than n words.

Got long line: üks kaks kolm neli
Got long line: uno due tre quattro

** 6 ** Generator closed

Traceback (most recent call last):
  File "/home/werner/python/Priming.py", line 41, in <module>
    gen.send("ichi ni san")
StopIteration
```

We also illustrated the method **close()** which closes the generator: attempting to send more information to it raises an exception.

6.4 Pipes of generators and coroutines

Generators' and coroutines' ability to receive and send information allows us to built out of them configurable "pipes" of actions where output of one of them is input of the next. It somehow resembles streams with their intermediate and terminal operations in Java.

Let us first consider such a pipe of coroutines. They will receive consecutive pieces of information (using **yield** expressions) from the previous coroutine and, after some kind of processing, "push" them, (using **send**) to the next. Of course, we have to start from generating initial stream of data which will be pushed into the pipe, and eventually end the whole procedure with a "sink" — something that consumes data but doesn't pass it any further.

Let us consider and example. In the program below

Listing 32 ABW-PushPipe/PushPipe.py

```
#!/usr/bin/env python

import functools

```

```python
  # decorator priming coroutines
def coroutine(f):
    @functools.wraps(f)
    def primed(*args,**kwargs):
        corou = f(*args,**kwargs)
        next(corou)
        return corou
    return primed

  # passes only n items
@coroutine
def limi(n, target):
    while True:
        item = (yield)
        if n > 0:
            print('lim'+str(item), end=' ')
            target.send(item)
        n -= 1

  # applies consumer, passes everything unchanged
@coroutine
def peek(consumer, target):
    while True:
        item = (yield)
        consumer(item)
        target.send(item)

  # passes only what meets predicate
@coroutine
def filt(predicate, target):
    while True:
        item = (yield)
        if predicate(item):
            print('fil'+str(item), end=' ')
            target.send(item)

  # passes everything mapped by func
@coroutine
def mapp(func, target):
    while True:
        item = (yield)
        print('map'+str(item), end=' ')
        target.send(func(item))

  # pushes data to pipe
def gen(source, target):
    for e in source: target.send(e)

  # sink: appends everything to list
@coroutine
```

```
55 def toList(lst):
56     while True:
57         item = (yield)
58         print("sink" + str(item), end=' ')
59         lst.append(item)
60 
61 
62 src = (x for x in range(1, 7))     # for initial generator
63 pred = lambda n: not n%2           # for filtering
64 func = lambda x: 'Res'+str(x)      # for mapping
65 cons = lambda x: print('peek'+str(x), end=' ')  # for peek
66 lst = []                           # for toList
67 
68   # either this...
69 crMapp = mapp(func, toList(lst))
70 crFilt = filt(pred, crMapp)
71 crLimi = limi(4, crFilt)
72 crPeek = peek(cons, crLimi)
73 gen(src, crPeek)                   # starting the pipe
74 
75   # or this
76 #gen(src, peek(cons, limi(4, filt(pred, mapp(func, toList(lst))))))
77 
78 print('\nResulting list:', lst)
```

- The **gen** function iterates over an iterable source (defined on line 62) and sends elements, one by one, to target. In the example, this target will be the **peek** coroutine.[1]
- The **peek** coroutine applies the **consumer** function (defined on line 65) to all elements, but otherwise sends them intact to the next coroutine, which is **limi**.
- The **limi** coroutine passes only the first n elements (4 in the example), ignoring all the remaining, to **filt**.
- The **filt** coroutine takes a predicate (defined on line 63) and rejects all elements which do not meet it; those which do are passed to **mapp**.
- The **mapp** coroutine applies the **func** function (defined on line 64) to all elements and sends the results to **toList**.
- The **toList** coroutine is the sink: it just appends all arriving elements to a list, which was passed to it as the argument.

We can compose one chain of operations as on line 76, or to make it more clear, split the problem into individual operations, as on lines 69-73 (which should be read backwards).

In both cases, the result printed is (the first two lines are in fact one line)

```
peek1 lim1 peek2 lim2 fil2 map2 sinkRes2 peek3 lim3
    peek4 lim4 fil4 map4 sinkRes4 peek5 peek6
Resulting list: ['Res2', 'Res4']
```

[1] All coroutines are primed by the **coroutine** decorator.

(as we can see, elements 5 and 6 were pushed to **peek**, although they were not needed because of **limi**.)

Let us now consider the same chain of operations, but using generators “pulling” data from the previous ones.

Listing 33 ABX-PullPipe/PullPipe.py

```
#!/usr/bin/env python

import functools

  # passes only n items
def limi(n, source):
    for _ in range(n):
        item = next(source)
        print('lim'+str(item), end=' ')
        yield item

  # applies consumer, passes everything unchanged
def peek(consumer, source):
    for item in source:
        consumer(item)
        yield item

  # passes only what meets predicate
def filt(predicate, source):
    for item in source:
        if predicate(item):
            print('fil'+str(item), end=' ')
            yield item

  # passes everything mapped by func
def mapp(func, source):
    for item in source:
        print('map'+str(item), end=' ')
        yield func(item)

  # pushes data to pipe
def gen(source, target):
    for e in source:
        target.send(e)

  # sink: appends everything to list
def toList(source):
    lst = []
    for item in source:
        print("sink" + str(item), end=' ')
        lst.append(item)
    return lst

```

```
44
45 src = (x for x in range(1, 7))      # for initial generator
46 pred = lambda n: not n%2            # for filtering
47 func = lambda x: 'Res'+str(x)       # for mapping
48 cons = lambda x: print('peek'+str(x), end=' ')  # for peek
49 lst = []                            # for toList
50
51   # either this
52 grPeek = peek(cons, src)
53 grLimi = limi(4, grPeek)
54 grFilt = filt(pred, grLimi)
55 grMapp = mapp(func, grFilt)
56 res = toList(grMapp)                # starting the pipe
57
58   # or this
59 #res = toList(mapp(func, filt(pred, limi(4, peek(cons, src)))))
60
61 print('\nResulting list:', res)
```

This time we start from the other end:

- The **toList** function collects all elements in the list, pulling data from **mapp**.
- The **mapp** generator applies the **func** function (defined on line 47) to all elements pulled from the source, which is **filt**.
- The **filt** generator takes a predicate (defined on line 46) and rejects all elements which do not meet it; it pulls data from **limi**.
- The **limi** generator passes only the first `n` elements (4 in the example), ignoring all the remaining; it pulls data from **peek**.
- The **peek** generator applies the **consumer** function (defined on line 48) to all elements, but otherwise resends them intact; it pulls data from the source defined as a generator on line 45.

Again, we can compose one chain of operations as on line 59, or to make it more clear, split the problem into individual operations, as on lines 52-56 (which should again be read backwards).

In both cases, the result printed is (the first two lines are in fact one line)

```
peek1 lim1 peek2 lim2 fil2 map2 sinkRes2 peek3 lim3
    peek4 lim4 fil4 map4 sinkRes4
Resulting list: ['Res2', 'Res4']
```

(as we can see, this time elements 5 and 6 were *not* pulled by **peek**, because **limi** demanded only 4 elements.)

6.5 Some tools from itertools

The ***itertools*** module contains several tools that one can use when dealing with iterables. One important example was already described, namely the **accumulate** function (see sec. 5.4.4, p. 82).

Other useful functions from this module are **dropwhile** and **takewhile**. They both take a predicate and an iterable and return an iterable. The first just *ignores* elements from the beginning of the input iterable as long as they meet the predicate, while the second includes *only* these elements:

```
>>> from itertools import dropwhile, takewhile
>>> ls = [1, 2, 3, 4, 5, 6]
>>>
>>> drop = dropwhile(lambda x: x < 4, ls)
>>> type(drop)
<class 'itertools.dropwhile'>
>>> drop
<itertools.dropwhile object at 0x7f8d1050ddc0>
>>> list(drop)
[4, 5, 6]
>>>
>>> take = takewhile(lambda x: x < 5, ls)
>>> type(take)
<class 'itertools.takewhile'>
>>> take
<itertools.takewhile object at 0x7f8d1050e080>
>>> list(take)
[1, 2, 3, 4]
```

Note that the **takewhile** function may be useful when dealing with potentially infinite iterables and we need to ensure that the iteration will stop at some point.

Somewhat related to the two previous function is the **islice** function. It allows us to select from an iterable only some of the yielded elements. Its syntax is similar to that of **range** (and also to slicing)

```
itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])
```

The function returns an iterable which will return only items selected by specifying start, step and step.

Invoking it with one argument (besides the iterable), it is equivalent to

```
itertools.islice(iterable, 0, stop, 1)])
```

i.e., first stop elements will be selected (we could say *from index 0 inclusive to index* stop *exclusive,* but of course the input iterable is not necessarily indexable).

With two arguments (besides the iterable), it is equivalent to

```
itertools.islice(iterable, start, stop)])
```

i.e., first start elements will be skipped, and the next stop−start selected; stop may be **None**, which means *until the input iterable is exhausted.* (we could thus say *from index* start *inclusive to index* stop *exclusive*).

In both previous cases, the step is assumed to be 1, i.e., all elements from a given range are selected. As is the case for **range** (and slices), setting step to a value greater than 1, say n, selects every n-th item; we then have to use three arguments, although both may be **None** (for start it's equivalent to 0, for stop – to *to the end*).

For example:

Listing 34 ABM-Islice/Islice.py

```
#!/usr/bin/env python

from sys import maxsize
from itertools import islice

iterable = '0123456789'

for e in islice(iterable, 5):
    print(e, end=' ')   # 0 1 2 3 4
print()

for e in islice(iterable, 2, 8):
    print(e, end=' ')   # 2 3 4 5 6 7
print()

for e in islice(iterable, None, None, 3):
    print(e, end=' ')   # 0 3 6 9
print()


  # 'almost infinite' sequence of primes
iterable = (k for k in range(2, maxsize)
            if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
  # first 10 primes
for e in islice(iterable, 10):
    print(e, end=' ')   # 2 3 5 7 11 13 17 19 23 29
print()

  # we have to recreate the iterable, because
  # generators cannot be 'rewound'!
iterable = (k for k in range(2, maxsize)
            if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
 # 100-th to 105-th primes
for e in islice(iterable, 100, 106):
    print(e, end=' ')   # 547 557 563 569 571 577
print()

iterable = (k for k in range(2, maxsize)
            if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
 # every third prime number from 1000-th to 1021-th
for e in islice(iterable, 1000, 1022, 3):
    print(e, end=' ')   # 7927 7949 7993 8017 8059 8087 8101 8123
print()
```

The **count** is another useful function. It is similar to **range**, but produces always infinite sequence starting from start by step; both stop and step do not have to be integer, as they don't correspond to "indices"

```
itertools.count(start=0, step=1)
```

For example

```
>>> from itertools import count, takewhile, islice
>>>
>>> cnt = count()
>>> type(cnt)
<class 'itertools.count'>
>>> list( takewhile(lambda x: x < 10, cnt) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> list( islice(count(1.5, 2.5), 100, 106) )
[251.5, 254.0, 256.5, 259.0, 261.5, 264.0]
```

Keep in mind that **count** produces infinite stream of items, so we always have to ensure that only finite number of elements will remain, for example, as in the example above, by using **takewhile** or **islice**.

As the last function from the ***itertools*** module, we will briefly describe the **groupby** function

```
itertools.groupby(iterable, key=None)
```

It takes an iterable and a key function, similar to what we use in the **sorted**/**sort** functions. This key function should yield, for elements of the input iterable, values which are then used as the keys by which the elements are to be grouped (by default this is just identity function). Elements corresponding to the same key value that should fall into the same group must be adjacent; therefore, very often, the iterable must be first sorted with same key function.

The function returns an iterable yielding pairs with a group key as the first element and an iterable of this group's elements as the second one.

For example

Listing 35 ABN-Groups/Groups.py

```
#!/usr/bin/env python

from itertools import groupby

lst = ['frog', 'goat', 'France', 'Greg', 'floor']

  # key function: by first letter case insensitively
keyfun = lambda s: s.lower()[0]

  # sorting
lst.sort(key=keyfun)

print(lst, end='\n\n') # ['frog', 'France', 'floor', 'goat',
 ↪  'Greg']

groups = groupby(lst, key=keyfun)
```

```
print(type(groups), end='\n\n')  # <class 'itertools.groupby'>

for k, v in groups:
    print(type(v))
    print(f"Group '{k}' -> {list(v)}\n")
```

The program prints

```
['frog', 'France', 'floor', 'goat', 'Greg']

<class 'itertools.groupby'>

<class 'itertools._grouper'>
Group 'f' -> ['frog', 'France', 'floor']

<class 'itertools._grouper'>
Group 'g' -> ['goat', 'Greg']
```

Chapter 7

Collections

We have already been using collections, like lists and tuples: here, we will put together their features and say a few words about other collections, that we haven't covered yet.

It is a very characteristic feature of the Python language that it already supports several types of collections which in other languages are rather a part of their standard libraries (for example, lists, dictionaries (maps), sets or tuples). Nevertheless, there are others, which also in Python are implemented not in the language itself but are available as standard packages/modules.

Generally, *collection* is something which fulfills the Collection protocol, and therefore

- implements the Container protocol, i.e., responds to the in and not in operators (it will, if it has the __contains__ method).
- is iterable — implements the Iterable protocol which requires that the iter function will work (it will, if it has the __iter__ method).
- implements the Sized protocol, so the len function works (it will, if it has the __len__ method).

7.1 Sequential collections

Sequential collections must be, of course, collections, but, additionally, be Reversible, i.e., respond to functions index, count, reversed (they will, if the __getitem__ and __reversed__ methods are implemented).[1]

Sequential collections can be mutable (as list) or immutable (as tuple). If a collection is to be mutable, it has to fulfill also the MutableSequence protocol, what means that it should react to functions insert, append, extend, pop, remove (so in addition to those already mentioned, it should have __setitem__, __delitem__ and __add__ methods).

7.1.1 Non-modifying operations

Non-modifying operations on sequential collections are illustrated below (we use lists, tuples and strings as examples). They can be used for both mutable and immutable collections.

in, not in

```
>>> lst = [2, 3, 4, 5, 6, 7, 8]
>>> k = 7
>>> k in lst
True
>>> k+2 not in lst
True
>>> k+2 in lst
False
```

[1] Contrary to seq[::-1], the reversed(seq) function (which calls __reversed__) doesn't make any copy — it just returns an iterator traversing the sequence backwards.

Usually, **in** and **not in** operators check if a given value is equal to the value of one of the elements of the collection. For objects of type **str** (and also for **bytes** and **bytearray**), also subsequences will be searched for, not necessarily a single element:

```
>>> s, s1, s2 = 'abcdef', 'cde', 'efg'
>>> s1 in s
True
>>> s2 in s
False
```

Concatenation

Concatenation of two sequences returns a new sequence: their type should be the same:

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> 'abc' + 'def'
'abcdef'
>>> (1, 2) + [3, 4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
```

Concatenation is *not* supported by objects of type **range**.
Concatenating a sequence several times with itself can be expressed by "multiplication":

```
>>> [1, 2]*3
[1, 2, 1, 2, 1, 2]
>>> 4*'abc'
'abcabcabcabc'
```

One has to be careful here and pay attention to whether the "replicated" object is, or is not mutable. That's because what is replicated are *references.* There is no problem, if they refer to immutable objects. However, if the object is mutable, then its modifications apply also to the objects referred to by the remaining copies of the reference, simply because they are actually the same object:

```
>>> ls = [1, [2, 3]]
>>> ls
[1, [2, 3]]
>>> ls2 = 2*ls
>>> ls2
[1, [2, 3], 1, [2, 3]]
>>> ls[1][1] = 9
>>> ls2
[1, [2, 9], 1, [2, 9]]
>>> id(ls[1]), id(ls2[1]), id(ls2[3])
(140148710923968, 140148710923968, 140148710923968)
```

Indexing and slicing

Indexing and slicing have been already described — see sec. 4.5, p. 43. Note that even ranges can be indexed and sliced, although it may be confusing and we have to be aware which arguments refer to values and which to indices. For example,

```
>>> r = range(10, 29, 3)
>>> r
range(10, 29, 3)
>>> type(r)
<class 'range'>
>>> list(r)
[10, 13, 16, 19, 22, 25, 28]
>>>
>>> q = r[1:4:2]
>>> q
range(13, 22, 6)
>>> type(q)
<class 'range'>
>>> list(q)
[13, 19]
>>>
>>> s = r[::-1]
>>> s
range(28, 7, -3)
>>> type(s)
<class 'range'>
>>> list(s)
[28, 25, 22, 19, 16, 13, 10]
```

We start with r which represents the arithmetic progression starting with the *value* 10, up to, but not including, value 29 and with 3 as the common difference; we get the sequence

$$[10, 13, 16, 19, 22, 25, 28]$$

Now we take the [1:4:2] slice — here, the numbers refer to *indices,* not values. Therefore, we get a sequence starting from element with index 1, which has value 13, up to, but not including, element with index 4, which is 22, and with the step of indices 2, that is we take every second element of the sequence. The result is therefore the sequence [13, 19] which corresponds to range(13, 22, 6).[1]

len

All collections must have well defined number of elements available by invoking the **len** function.

```
>>> ls = [1, 2, (3, 4, (5, 6, 7)), [(8, 9), (10,)]]
>>>
>>> len(ls)
4
>>> len(ls[2])
3
>>> len(ls[2][2])
3
>>> len(ls[3][1])
1
```

[1] Of course, **range(13,20,6)** or **range(13,25,6)** would represent exactly the same sequence of numbers.

min, max
Finding minimum or maximum element is possible when elements are comparable; for example, we can compare tuples with tuples, but not numbers with tuples. The < operator will be used for comparisons, so it has to have well defined meaning for any two elements of the sequence:

```
>>> ls = [(1,), (2,3)]
>>> min(ls)
(1,)
>>> ls = [1, (2,3)]
>>> min(ls)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'tuple' and 'int'
```

index(x[, i[, j]])
returns the index of the first occurrence of x in the sequence (or in its subsequence starting at index i up to, but not including, element with index j). If such element is not found, the **ValueError** exception is raised. Not all sequences support additional arguments (i and j):

```
>>> r = range(10, 31, 3)
>>> list(r)
[10, 13, 16, 19, 22, 25, 28]
>>> r.index(19)
3
>>> r.index(19, 1, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range.index() takes exactly one argument (3 given)
```

count(x)
returns number of occurrences of x in the sequence. Works also for ranges:

```
>>> r = range(10, 31, 3)
>>> r.count(19)
1
>>> r.count(20)
0
```

7.1.2 Modifying operations

Mutable collections support more operations — those which do modify them and therefore cannot be used with immutable ones.

Modifying elements

```
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7]
>>> ls[2] = 'two'
>>> ls
[0, 1, 'two', 3, 4, 5, 6, 7]
```

We can also assign to a slice of a collection, but then the right-hand side will be treated as an iterable and its elements will be inserted one by one:

```
>>> ls = [0, 1, 2, 3]
>>> ls[1:3] = 'one two'
>>> ls
[0, 'o', 'n', 'e', ' ', 't', 'w', 'o', 3]
```

If what we mean is a string as a whole, we can make it a sole element of a tuple (note the comma):

```
>>> ls = [0, 1, 2, 3]
>>> ls[1:3] = 'one two',   # note the comma
>>> ls
[0, 'one two', 3]
```

However, if a slice has step different than one, then right-hand side must be an iterable with exactly as many elements as there are elements in the slice:

```
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ls[::3] = 'zero', 'three', 'six', 'nine'
>>> ls
['zero', 1, 2, 'three', 4, 5, 'six', 7, 8, 'nine']
>>>
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ls[::3] = 'zero', 'three', 'six'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of
size 3 to extended slice of size 4
```

Adding elements

New elements can be added to a mutable sequence by method **insert** which takes an index specifying the location where a new element is to be inserted, and its value. If the length of the sequence is L, then indices greater than L are equivalent to L (no exception raised!). Indices may be negative with usual interpretation: -1 corresponds to the last element and -L to the first. Again, indices smaller than -L are equivalent to -L (no exception raised):

```
>>> l1 = ['b', 'd']
>>> l2 = ['b', 'd']
>>> l3 = ['b', 'd']
>>> l4 = ['b', 'd']
>>> l5 = ['b', 'd']
>>> l1.insert(2, 'e')
>>> l2.insert(3, 'e')
>>> l1, l2
(['b', 'd', 'e'], ['b', 'd', 'e'])
>>> l3.insert(-2, 'a')
>>> l4.insert(-3, 'a')
>>> l3, l4
(['a', 'b', 'd'], ['a', 'b', 'd'])
```

```
>>> l5.insert(-1, 'c')
>>> l5
['b', 'c', 'd']
```

The **append** method appends a single element (which may be an iterable) to the collection:

```
>>> ls = [1, 'a']
>>> ls.append(['X', 'Y'])
>>> ls
[1, 'a', ['X', 'Y']]
```

On the other hand, the **extend** method takes an iterable and adds its elements individually to the sequence:

```
>>> ls = [1, 'a']
>>> ls.extend('one')
>>> ls
[1, 'a', 'o', 'n', 'e']
>>>
>>> ls = [1, 'a']
>>> ls.extend(('one',))  # note the comma
>>> ls
[1, 'a', 'one']
```

Instead of calling the **extend** method, we can use the **+=** operator:

```
>>> ls = [1, 'a']
>>> ls += 'one'
>>> ls
[1, 'a', 'o', 'n', 'e']
>>>
>>> ls = [1, 'a']
>>> ls += 'one',         # note the comma
>>> ls
[1, 'a', 'one']
```

Removing elements

The **del** statement can be used to delete chosen elements of a sequence (or the whole sequence; **del** may delete any object). We can delete a single element, or a slice:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> del ls[2]
>>> ls
['a', 'b', 'd', 'e', 'f', 'g', 'h']
>>> del ls[1:3]
>>> ls
['a', 'e', 'f', 'g', 'h']
>>> del ls[::2]
>>> ls
['e', 'g']
```

To remove elements of a sequence, we can also use the **remove** method, which takes a value and removes the first element of a sequence with this value, or raises **ValueError** exception if such element has not been found:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> ls.remove('d')
>>> ls.remove('g')
>>> ls
['a', 'b', 'c', 'e', 'f', 'h']
>>>
>>> ls.remove('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Finally, we can use **pop** function. It takes the index of an element to be removed (it has a default value -1, what corresponds to the the *last* element) and not only removes it from the sequence, but also returns its value:

```
>>> ls = ['a', 'b', 'c', 'd', 'e']
>>> x = ls.pop()
>>> x, ls
('e', ['a', 'b', 'c', 'd'])
>>>
>>> y = ls.pop(1)
>>> y, ls
('b', ['a', 'c', 'd'])
>>>
>>> z = ls.pop(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

Copying

The **copy** function returns a *shallow* copy of the collection:

```
>>> ls = [1, [2, 3], 'a']
>>> lc = ls.copy()
>>> ls[1][1] = 9
>>> lc
[1, [2, 9], 'a']   # lc also modified
```

We can also replace the collection by n shallow copies of itself using the ***=** operator (shallow, because what is copied are references, not objects):

```
>>> ls = [1, [2, 3]]
>>> ls *= 3
>>> ls
[1, [2, 3], 1, [2, 3], 1, [2, 3]]
>>> ls[1][1] = 9
>>> ls
[1, [2, 9], 1, [2, 9], 1, [2, 9]] # all copies modified
```

Reversing and clearing

Finally, the **clearclear (function)** function clears the collection: makes it empty, but doesn't delete it. The **reverse reverse (function)** function reverses the order of all elements of the collection *in situ* and returns **None**:

```
>>> ls = [1, [2, 3], 'a']
>>> x = ls.reverse()        # reverse in situ
>>> print(x)
None
>>> ls
['a', [2, 3], 1]
>>> ls.clear()
>>> ls
[]
```

7.1.3 *bytes* objects

Objects of type **bytes** represent *immutable* sequences of bytes. They are implemented as sequences of one-byte unsigned integers in the range $[0, 255]$. Literals of this type look as literals of strings, with the letter 'b' (or 'B') before the opening quote (as for strings, quotes can be single or double; also the "tripled" version is supported). Apart for 'b', we can additionally use the 'r' (or 'R') prefix to get the "raw" interpretation, in which backslashes are not special and stand for themselves.

When writing **bytes** literals, we can use just single characters for values corresponding to ASCII characters. For values greater then 126 ($= 7E_{16}$) or those corresponding to control characters, i.e., smaller than 32 ($= 20_{16}$), the `\xhh` notation is used, where `hh` stands for exactly two-digit hexadecimal number: each digit corresponding to one nibble of a byte.[1] This is also the form in which **bytes** objects are displayed. For example, in `b'B\xc4\x85k'` the 'B' stands for 66 (as 66 is the Unicode code point of this letter), 'k' at the end stands for 107 (code point of 'k'). Two middle bytes correspond to 196 and 133 — taken together they happen to be the UTF-8 two-byte code of the Polish letter 'ą'.

```
>>> bąk = b'B\xc4\x85k'
>>> bąk
b'B\xc4\x85k'
```

Objects of type **bytes** sometimes *do* represent texts, and not some binary data. If this is the case, we can use methods allowing us to transform bytes into strings and *vice versa.* For example, to transform **bytes** into string, we can pass **bytes** to the constructor of **str** specifying the encoding[2]

```
>>> bąk = b'B\xc4\x85k'   # this is 'bąk' in UTF-8
>>> type(bąk)
<class 'bytes'>
>>> st = str(bąk, 'utf_8')
>>> type(st)
<class 'str'>
>>> st
'Bąk'
```

and in a similar way in the opposite direction

[1] Notations `\uhhhh` and `\Uhhhhhhhh` for 16- and 32-bit Unicode code points is supported for strings only.

[2] Many names of encodings contain the underscore character (_) like 'utf_8', 'koi8_u', etc. Sometimes the form with a dash is also acceptable as an alias (e.g., 'uft-8'), but using aliases should be avoided, as it may lead to slightly worse efficiency.

```
>>> bt = bytes('Bąk', 'utf_8')
>>> type(bt)
<class 'bytes'>
>>> bt
b'B\xc4\x85k'
```

Objects of type **bytes** are really sequences of integers; we can see it by passing such object to the constructor of **list**

```
>>> bt = bytes('Żółć', 'utf_8')
>>> list(bt)
[197, 187, 195, 179, 197, 130, 196, 135]
```

or by creating **bytes** object from any iterable yielding a sequence of integers

```
>>> bt = bytes([197, 187, 195, 179, 197, 130, 196, 135])
>>> str(bt, 'utf_8')
'Żółć'
```

The **fromhex** function may be used to create a **bytes** object from a string containing numbers in hexadecimal notation: 2 hex-digits for a byte (one digit per a nibble). The string may contain spaces for better readability — they will be ignored:

```
>>> bt = bytes.fromhex('42 c485 6b')
>>> str(bt, 'utf_8')
'Bąk'
```

The opposite operation, from a **bytes** object to string with numbers in hexadecimal notation is also possible by means of the **hex** method (we can pass a separator to be used between numbers):

```
>>> bt = bytes('Żółć', 'utf_8')
>>> type(bt)
<class 'bytes'>
>>> bt.hex()
'c5bbc3b3c582c487'
>>> bt.hex('-')
'c5-bb-c3-b3-c5-82-c4-87'
```

Usually **bytes** object are used to store binary data, not just strings. This data is very often read from a file, or written to a file (see sec. ??, p. ??):

```
>>> bt = bytes('Żółć', 'utf_8')
>>> with open('bytes.dat', 'wb') as f:
...     f.write(bt)
...
8
>>> with open('bytes.dat', 'rb') as f:
...     bb = f.read()
...
>>> str(bb, 'utf_8')
'Żółć'
```

Note that files has to be open in the binary mode. The **write** function returns the number of bytes written. Sometimes the logic of the program requires that the bytes are written/read one by one

```
1   >>> bt = bytes('Żółć', 'utf_8')
2   >>> with open('bytes.dat', 'wb') as f:
3   ...     for i in range(len(bt)):
4   ...         f.write(bt[i:i+1])
5   ...
6   >>> with open('bytes.dat', 'rb') as f:
7   ...     bt = bytes()
8   ...     while (b := f.read(1)):
9   ...         bt += b
10  ...
11  >>> str(bt4, 'utf_8')
12  'Żółć'
```

Note that on line 4 we had to use a one-character slice, because its type is **bytes** — just `bt[i]` would be an integer.
In line 8 we used the **read** function with an argument 1 — it says that only one byte is to be read. Also the *walrus* assignment expression was used here: it works, because when the end of file is detected, **read** returns an empty string, which is interpreted as **False**.
Line 9 is, of course, very ineffective: **bytes** object are immutable, so `bt += b` creates new objects in each iteration.

Objects of type **bytes** are immutable sequences and hence they have all the non-modifying methods described in sec. 7.1.1, p. 118.
They are also similar to strings, and therefore also methods of the **str** class (described in sec. 8, p. 158) may be used for **bytes**. One has to remember, though, that analogous method of the **str** class which require arguments of type **str**, here, for **bytes**, need arguments of byte types, i.e., **bytes** or **bytearray**:

```
>>> bt = b'abcdef'
>>> a = bt.replace(b'c', b'C')
>>> type(a), a
(<class 'bytes'>, b'abCdef')
>>> b = bt.replace('c', 'C')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a bytes-like object is required, not 'str'
```

7.1.4 *bytearray* objects

Objects of type **bytearray**, as those of type **bytes**, represent sequences of bytes, i.e., integer numbers in the range $[0, 255]$, but they are, unlike **bytes**, mutable.

There are no literals of this type. However, the **__str__** method for objects **bytearray** returns a string in the format as for **bytes** objects, e.g., numbers from the range $[0, 126]$ are printed as the characters corresponding to them, and the remaining numbers are displayed in the `\xhh` form, where `hh` denotes exactly two hexadecimal digits

```
>>> bytearray([30, 31, 32, 33, 125, 126, 127, 128])
bytearray(b'\x1e\x1f !}~\x7f\x80')
```

Objects **bytearray** can be created in several ways:

- Not passing any arguments to the constructor — an empty array is then created:

  ```
  >>> bytearray()
  bytearray(b'')
  ```

- Passing an integer number to the constructor — an array of this size will be created and filled with zeros:

  ```
  >>> bytearray(8)
  bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
  ```

- Passing to the constructor an iterable providing integer numbers from the range $[0, 255]$ — an array containing these bytes will be created:

  ```
  >>> bytearray(x for x in range(65, 91))
  bytearray(b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
  ```

- Passing to the constructor a string and, as the second argument, an encoding — the string is then converted to a sequence of bytes appropriate for a given encoding:

  ```
  >>> bytearray("ABC Żółć DEF", 'utf_8')
  bytearray(b'ABC \xc5\xbb\xc3\xb3\xc5\x82\xc4\x87 DEF')
  ```

- Passing to the constructor an object possessing a binary buffer, as, for example, **bytes** — the buffer is then copied:

  ```
  >>> bytearray(b'abc')
  bytearray(b'abc')
  ```

- Calling the static **fromhex** method of the **bytearray**, passing a string containing bytes in the `hh` format — two hexadecimal digits for each byte (for better readability, white characters may also be used in the string, they will be ignored):

  ```
  >>> bytearray.fromhex('CAFE BABE')
  bytearray(b'\xca\xfe\xba\xbe')
  ```

Objects of type **bytearray** are mutable sequences and hence they have all the methods described in both sec. 7.1.1, p. 118 and sec. 7.1.2, p. 121.
As for **bytes**, objects of type **bytearray** have methods analogous to those of the **str** type (see sec. 8, p. 158), but where methods of the **str** have arguments of type string, here, for **bytearray** they have to be byte objects,i.e., **bytes** or **bytearray**:

```
>>> ba = bytearray(b'abcdef')
>>> ba.endswith(b'def')
True
>>> ba.endswith('def')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: endswith first arg must be bytes or a tuple of bytes, not str
```

7.2 Hashable objects

Python objects can be hashable or not. This is an important distinction if we want to use them as elements of sets and/or keys of dictionaries.

In Java, both dictionaries (called maps in Java) and sets[1] come in two flavors: with implementation based on a red-black tree – **TreeMap**, **TreeSet** – or on an associative array (hashing) – **HashMap**, **HashSet**. In Python, however, standard sets and dictionaries are based on hashing only. This is slightly faster — $\mathcal{O}(1)$ vs. $\mathcal{O}(\log(n))$ for the tree implementation — and doesn't require elements to be comparable. But, as we remember from Java, the associative-array implementation requires a function which calculates, for any given object, its **hash** (or hash code): an integer which will determine the index of a bucket where this object is to be placed (or searched for). [2] To resolve possible conflicts, when for more than one different objects the index comes out the same, a function (a bi-predicate) is needed which will be able to determine if two objects are, or are not, equal.

For this to work, the hash of an object should be constant. Normally, however, the hash is calculated based on the value(s) associated with the object — that means that these objects should be immutable. And indeed, immutable objects, like numbers and strings, do have well defined **__hash__** method (which is implicitly called by the built-in **hash** function):

```
>>> hash(17), (17).__hash__()
(17, 17)
>>> hash('Alice'), 'Alice'.__hash__()
(-5185723981070768863, -5185723981070768863)
```

Numbers and strings represent only one value on which their hash codes depend. What about collections, which are aggregates of values? Only immutable collections (like tuples) can be hashable — and only when they don't contain mutable, and therefore not hashable, elements:

```
>>> tp = 1, 'abc', (2, 'de')      # all elements hashable
>>> tp
(1, 'abc', (2, 'de'))
>>> hash(tp)
-1823045235319932986
>>>
>>> tq = 1, 'abc', [2, 'de']      # list is mutable
>>> tq
(1, 'abc', [2, 'de'])
>>> hash(tq)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Generally, Python objects are hashable if they have hash values (available by means of the **__hash__** method) which never change. They also have to have the **__eq__** method, such that

if two object are considered equal by **__eq__** method, their hashes must be identical.

[1] Actually, the implementation of sets is usually backed by maps (dictionaries).

[2] https://en.wikipedia.org/wiki/Hash_table

Since hashes of objects are based on their values, mutable collections[1] cannot guarantee immutability of their hash value. The same applies to immutable collections,[2] but containing mutable elements.

Note that objects of user-defined classes *are* hashable by default, as they inherit __hash__ and __eq__ methods from the class **object**. However, their default implementations is based solely on the identifiers id of objects (so effectively on their addresses). Therefore, to get something more meaningful for our custom types, we have to provide implementations of __hash__ and __eq__ methods ourselves.

7.3 Sets

Sets represent *unordered* collection of *distinct*[3] hashable objects. The implementation is based on hash values of objects, so elements of a set must be hashable, and consequently, immutable.

Being collections, sets support **in** and **not in** operators as well as **len** function. They are iterable, although the order of elements yielded by iteration is essentially not predictable. They are *not* sequential, though. Therefore, indexing and slicing is not supported by sets.

There are two built-in types of sets in Python: **set** and **frozenset**. In both cases, elements must be hashable (immutable), but objects of type **set** themselves *are* mutable — we can add elements to sets, or remove them from sets. Therefore, sets are not hashable and cannot be elements of other sets:

```
>>> st = {'1', 2, '3', (4, '5')}
>>> id(st)
139913779179840
>>> st.add(6)
>>> id(st)
139913779179840  # same object (modified)
>>> st
{2, 6, '3', '1', (4, '5')}  # order unpredictable
>>> newset = {st, '7', 8}   # set as element of set?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Objects of type **frozenset**, however, are immutable and hashable. When created, no elements can be added or removed from them. Consequently, they may be used as elements of other sets, or as keys in dictionaries:

```
>>> st = frozenset(['1', 2, '3', (4, '5')])
>>> newset = {st, '7', 8}
>>> newset
{8, '7', frozenset({'3', '1', 2, (4, '5')})}
```

Non-empty sets may be created by passing to the constructor an iterable — possible repetitions will be automatically ignored. Objects of the **set** type (but not **frozenset**) may be also created by using a literal: an enclosed in braces list of comma-separated

[1] These are, e.g., **list**, **dict**, **set**, **bytearray**.

[2] Like **int**, **float**, **decimal**, **complex**, **bool**, **str**, **tuple**, **range**, **frozenset**, **bytes**.

[3] For any two elements of a set, a and b, the value of the expression a == b must be **False**.

elements.[1] Of course, in both cases, all elements which are to be placed in a set must be hashable:

```
>>> s1 = set()
>>> s2 = {1, '2', (3, '4')}  # only set, not frozenset
>>> s3 = set(int(x)**2 for x in [5, '6'])
>>> s4 = frozenset('789')
>>> s5 = frozenset(('789',)) # note the comma
>>> s1, s2, s3
(set(), {1, (3, '4'), '2'}, {25, 36})
>>> s4, s5
(frozenset({'8', '7', '9'}), frozenset({'789'}))
```

As for other collections, for sets we also can use **in** and **not in** operators and the **len** function:

```
>>> st = {1, '2', 3, '4'}
>>> len(st)
4
>>> 2 in st
False
>>> '2' in st
True
>>> '3' not in st
True
```

We can add (hashable) elements to sets (but not to **frozenset**s) using the **add** method; for removing elements we even have three methods:

- **remove(elem)** — removes element elem, and if there is no such element, raises the **KeyError** exception;
- **discard** — removes element elem, but *doesn't* raise any exception if there is no such element;
- **pop()** — removes one arbitrary element and returns it; raises **KeyError** exception if the set is empty.

For example

```
>>> st = {1, '2', 3, '4'}
>>> st.add(5)
>>> st.discard('6')  # there is no '6' but OK
>>> st.remove('2')
>>> st
{1, 3, 5, '4'}
>>> st.pop()
5
>>> st
{1, 3, '4'}
>>> st.remove(7)      # there is no 7 -> exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 7
```

[1] But empty braces would be interpreted as a literal of an empty *dictionary*, not of a set!

Sets can also be copied by using the **copy** method; this is a shallow copy, but normally it doesn't pose any problem, as elements are immutable anyway:

```
>>> A = {1, 2, 3, 4}
>>> B = frozenset(A)
>>> Ac = A.copy()
>>> Bc = B.copy()
>>> type(Ac), Ac
(<class 'set'>, {1, 2, 3, 4})
>>> type(Bc), Bc
(<class 'frozenset'>, frozenset({1, 2, 3, 4}))
```

Sets may be compared by invoking methods, or by using binary operators, the operands of which are sets (but it's possible, that one them is of type **set** and the other **frozenset**). They all return **True** or **False**.

Below, A and B denote sets.

- `A == B`, `A != B` — answer the question if A is (is not) the same as B; sets are equal, if $A \subset B$ and $B \subset A$ (there are no *methods* corresponding to these operators):[1]

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A == B, A != B
(False, True)
```

- `A.isdisjoint(B)` — answers the question, if intersection $A \cap B$ is empty (there is no operator corresponding to this method):

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {4, 5}
>>> A.isdisjoint(B), A.isdisjoint(C)
(False, True)
```

- `A.issubset(B)` or `A <= B` — answer the question, if $A \subset B$, that is if every element of A is also an element of B; in particular, this will be true, if `A == B`:

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A.issubset(B), B.issubset(A)
(False, True)
>>> A <= B, B <= A
(False, True)
```

- `A < B` — answers the question if $A \subset B$, but $A \neq B$, that is if every element of A belongs also to B, but `A != B` (there is no method corresponding to this operator):[2]

[1] Still, one *can* use “magic” methods **__eq__** and **__ne__**.

[2] Still, one *can* use the “magic” method **__lt__**.

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {1, 3}
>>> B < A, C < B
(True, False)
```

● `A.issuperset(B)` or `A >= B` — answer the question, if $A \supset B$, that is if every element of B is also an element of A; in particular, this will be true, if `A == B`:

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A.issuperset(B), B.issuperset(A)
(True, False)
>>> A >= B, B >= A
(True, False)
```

● `A > B` — answers the question if $A \supset B$, but $A \neq B$, that is if every element of B belongs also to A, but `A != B` (there is no method corresponding to this operator):[1]

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {1, 3}
>>> A > B, C > B
(True, False)
```

Note that subset and equality comparisons *do not* define a total order (it's possible that all $A \subset B$, $B \subset A$ and $A = B$ are false). Therefore, collections of sets cannot be sorted.

Another group of operators and methods correspond to operations on sets giving also a set as the result. Here again, we can use either methods invoked on a set with another set as the argument, or operators. There is a difference, however: for methods, the argument doesn't need to be a set, it can be any iterable yielding elements of a set — below, we denote this iterable by *elems. For operators, however, both operands must be sets, although not necessarily of the same type: one of them may be a **set**, but the other a **frozenset**. The type of the result will always be that of the left operand.

● `A.union(*elems)` or `A | set1 | set2 | ...` — return the union of sets:

```
>>> A = {1, 2, 3}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.union(BS)
>>> type(C), C
(<class 'set'>, {1, 2, 3, 6})
>>>
>>> D = A | BF    # result is set
>>> type(D), D
```

[1] Still, one *can* use the "magic" method **__gt__**.

```
(<class 'set'>, {1, 2, 3, 6})
>>>
>>> E = BF | A
>>> type(E), E    # result is frozenset
(<class 'frozenset'>, frozenset({1, 2, 3, 6}))
>>>
>>> F = A | BF | {7} | {'8'}  # operands must be sets
>>> type(F), F
(<class 'set'>, {1, 2, 3, 6, 7, '8'})
>>>
>>> G = A | [2, 3, 6]          # operand is not a set
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

- `A.intersection(*elems)` or `A & set1 & set2 & ...` — return the intersection of sets:

```
>>> A = {1, 2, 3}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.intersection(BS)
>>> type(C), C
(<class 'set'>, {2, 3})
>>>
>>> D = A & BF
>>> type(D), D
(<class 'set'>, {2, 3})
>>>
>>> E = BF & A
>>> type(E), E
(<class 'frozenset'>, frozenset({2, 3}))
>>>
>>> F = A & BF & {3, 4}
>>> type(F), F
(<class 'set'>, {3})
```

- `A.difference(*elems)` or `A - set1 - set2 - ...` — return the difference of sets:

```
>>> A = {1, 2, 3, 4}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.difference(BS)
>>> type(C), C
(<class 'set'>, {1, 4})
>>>
>>> D = A - BF
>>> type(D), D
```

```
(<class 'set'>, {1, 4})
>>>
>>> E = BF - A
>>> type(E), E
(<class 'frozenset'>, frozenset({6}))
>>>
>>> F = A - {3, 4}
>>> type(F), F
(<class 'set'>, {1, 2})
```

- `A.symmetric_difference(*elems)` or `A ^ set1 ^ set2 ^ ...` — return the symmetric difference of sets:

```
>>> A = {1, 2, 3, 4}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.symmetric_difference(BS)
>>> type(C), C
(<class 'set'>, {1, 4, 6})
>>>
>>> D = A ^ BF
>>> type(D), D
(<class 'set'>, {1, 4, 6})
>>>
>>> E = BF ^ A
>>> type(E), E
(<class 'frozenset'>, frozenset({1, 4, 6}))
>>>
>>> F = A ^ {3, 4}
>>> type(F), F
(<class 'set'>, {1, 2})
```

The methods and operators mentioned above can be used for both **sets** and **frozensets**. However, there are others which modify a set — these may be used *only* for sets of type **set**. They all return **None**.

- `A.update(*elems)` or `A |= set1 | set2 | ...` — modify A by adding given elements:

```
>>> A = {1, 2, 3, 4}
>>>
>>> id(A), A
(139927670937568, {1, 2, 3, 4})
>>>
>>> A.update({2, 4, 5, 6})
>>> id(A), A
(139927670937568, {1, 2, 3, 4, 5, 6})    # same id
>>>
>>> A |= {6} | {7}
>>> id(A), A
(139927670937568, {1, 2, 3, 4, 5, 6, 7}) # same id
```

- `A.intersection_update(*elems)` or `A &= set1 & set2 & ...` — modify A by keeping only the intersection of A with given elements:

```
>>> A = {1, 2, 3, 4}
>>>
>>> id(A), A
(139927690835584, {1, 2, 3, 4})
>>>
>>> A.intersection_update({2, 4, 5, 6})
>>> id(A), A
(139927690835584, {2, 4})
>>>
>>> A &= {4, 6} | {4, 7}
>>> id(A), A
(139927690835584, {4})
```

- `A.difference_update(*elems)` or `A -= set1 | set2 | ...` — modify A by keeping only elements of A that are not equal to any of the given elements:

```
>>> A = {1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
>>> id(A), A
(139927670938688, {1, 2, 3, 4, 5, 6, 7, 8, 9})
>>>
>>> A.difference_update({2, 4})
>>> id(A), A
(139927670938688, {1, 3, 5, 6, 7, 8, 9})
>>>
>>> A -= {1, 2} | {6, 7}
>>> id(A), A
(139927670938688, {3, 5, 8, 9})
```

- `A.symmetric_difference_update(*elems)` or `A ^= set1` — modify A so it contains only elements from A or set1, but not both:

```
>>> A = {1, 2, 3, 4, 5}
>>> B = A.copy()
>>> A.symmetric_difference_update({4, 5, 6, 7})
>>> B ^= {4, 5, 6, 7}
>>> A, B
({1, 2, 3, 6, 7}, {1, 2, 3, 6, 7})
```

- `A.clear()` — modify A so it becomes empty:

```
>>> A = {1, 2, 3, 4, 5}
>>> A
{1, 2, 3, 4, 5}
>>> A.clear()
>>> A
set()
```

7.4 Dictionaries

Mappings in Python correspond to what we know from Java as maps. This type is of the utmost importance in Python — as a matter of fact the implementation of the language itself is based on mappings.

There is only one built-in type of mapping, **dict** (from *dictionary*), which corresponds to what we know from Java as **LinkedHashMap**. This is a data structure holding as elements key/value pairs (called *entries* in Java and *items* in Python). The keys, as we know, have to be unique and hashable, as the implementation is hash-based (there is no a counterpart of **TreeMap** from Java).[1] Dictionaries "remember" the order in which its elements (key/value pairs) were inserted and return them in this order when iterated over.

We can create dictionaries in several ways:

- empty dictionary can be created by using the constructor without any arguments, or literal `{}` (empty braces):

```
>>> d1, d2 = dict(), {}
>>> type(d1), type(d2)
(<class 'dict'>, <class 'dict'>)
>>> len(d1), len(d2)
(0, 0)
```

- using literal form with enclosed in braces sequence of comma-separated pairs of keys and values, themselves separated by the colon:

```
>>> d = {'1': 'one', 'two': 2, 3: 3}
>>> d
{'1': 'one', 'two': 2, 3: 3}
```

- using keyword arguments in the constructor (their names, as strings, will become keys; of course, they have to be valid Python identifiers):

```
>>> d = dict(one=1, two='two', three='3')
>>> d
{'one': 1, 'two': 'two', 'three': '3'}
```

- from a dictionary comprehension:

```
>>> d = {x: y for x in range(10) if 20 < (y := x**2) < 85}
>>> d
{5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

- by passing to the constructor an iterable, each element of which is itself an iterable yielding exactly two values – the first will become the key, and the second its associated value; the iterable may be followed by keyword arguments:

```
>>> ls = [ (2, 3), ['7', 7], ((x, x**2) for x in range(4, 6)) ]
>>> d = dict(ls)
>>> d
{2: 3, '7': 7, (4, 16): (5, 25)}
>>>
>>> keys = [1, 2, 3]
>>> vals = ('one', 'two', 'three')
```

[1] Or **map** from C++. In C++, we do have **unordered_map** which is hash-based, but it's not "linked" — order of elements is essentially unpredictable.

```
>>> d1 = dict(zip(keys, vals))
>>> d1
{1: 'one', 2: 'two', 3: 'three'}
>>> d2 = dict(zip(keys, vals), four='four', six=6)
>>> d2
{1: 'one', 2: 'two', 3: 'three', 'four': 'four', 'six': 6}
```

- by passing another mapping to the constructor; again, possibly followed by keyword arguments:

```
>>> d = {'john': 'Doe', 'mary': 'Cooper'}
>>> d1 = dict(d, bill='Gatsby', kate='Coe')
>>> d1
{'john': 'Doe', 'mary': 'Cooper', 'bill': 'Gatsby', 'kate': 'Coe'}
```

- by using the class method **fromkeys(itarable[, val)** which returns a new dictionary with keys taken from the iterable and all values set to val (which is None, if omitted). All values are references to the same object, so it generally doesn't make much sense if it is mutable, as, e.g., a list:

```
>>> ls = ['john', 'mary']
>>> dict.fromkeys(ls)
{'john': None, 'mary': None}
>>> dict.fromkeys(ls, 0)
{'john': 0, 'mary': 0}
>>> d = dict.fromkeys(ls, [])     # john and mary refer
>>> d['john'].extend([1, 2, 3])  # to the same list!
>>> d['mary']
[1, 2, 3]
```

It may happen that when creating a dictionary, the same key will be used more than once with different values — then the last value prevails, although the original order is preserved:

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d1 = dict(d, three=33, one=11)
>>> d1
{'one': 11, 'two': 2, 'three': 33}
```

Let us summarize the operations supported by dictionaries (d denotes a dictionary):

- `len(d)` — returns the number of keys (or items, what's the same) in d.

- `list(d)` — returns the list of *keys* (not items!) of d:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> list(d)
['Joan', 'Bob', 'Tina']
```

- `d[key]` — returns the value associated with key in d; raises a **KeyError** if key is not in d (see also the **setdefault** method below):

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> d['Bob']
'Dylan'
>>> d['Beth']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Beth'
```

- `d[key]=val` — associates val with the key key; if d[key] already exists, val replaces the old value, if not, a new item is added to d:

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> d['Bob'] = 'Dylan'
>>> d['Tina'] = 'Turner'
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan', 'Tina': 'Turner'}
```

- `del d[key]` — removes the item with the given key from d; if such item doesn't exist, **KeyError** is raised (see also **pop** below):

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> del d['Bob']
>>> d
{'Joan': 'Baez'}
>>> del d['Tina']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Tina'
```

- `key in d`, `key not in d` — answer the question whether there is (there is no) item with key key in d:

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> 'Joan' in d
True
>>> 'Tina' in d
False
>>> 'Tina' not in d
True
```

- `iter(d)` — returns an iterator over *keys* of d; to iterate over items, i.e., key/value pairs (as 2-tuples), iterate over d.items():

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> for e in d:      # equivalent to 'for e in iter(d):'
...     print(e)     # and to 'for e in d.keys():'
...
Joan
Bob
Tina
```

```
>>>
>>> for e in d.items():
...     print(e)
...
('Joan', 'Baez')
('Bob', 'Dylan')
('Tina', 'Turner')
```

- `d.clear()` — makes the dictionary empty.

- `d.copy()` — returns a *shallow* copy of d:

```
>>> d = {'Joan': [1, 2], 'Bob': 'Turner'}
>>> c = d.copy()
>>> d['Joan'].append('X')
>>> d['Bob'] = 'Dylan'
>>> d
{'Joan': [1, 2, 'X'], 'Bob': 'Dylan'}
>>> c
{'Joan': [1, 2, 'X'], 'Bob': 'Turner'}
```

- `d.get(key[, default])` — returns the value associated with key if it exists; otherwise returns default (None if not specified). Never modifies the dictionary and never raises **KeyError**.

- `d.setdefault(key[, default])` — returns the value associated with key if it exists; otherwise adds a new item to d with key key and value default (None if not specified) and returns this value:[1]

```
>>> d = {'Joan': 'Baez'}
>>> d.setdefault('Joan', 'Turner')
'Baez'
>>> d.setdefault('Bob', 'Dylan')
'Dylan'
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan'}
```

- `d.pop(key[, default])` — if an item with key key exists, the value associated with this key is returned and the item is removed from d. Otherwise, default is returned and d is not modified — in this case default must be specified (or else a **KeyError** is raised):

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> d.pop('Bob')
'Dylan'
>>> d
{'Joan': 'Baez', 'Tina': 'Turner'}
>>> d.pop('Joe', 'No such element')
'No such element'
```

[1] Somewhat similar to **putIfAbsent** for maps in Java.

```
>>> d.pop('Joe')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Joe'
```

- `d.popitem()` — returns the last item[1] (key/value pair as a 2-tuple) and removes it from d; raises **KeyError** if d is empty:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> while (d):
...     print(d.popitem())
...
('Tina', 'Turner')
('Bob', 'Dylan')
('Joan', 'Baez')
```

- `reversed(d)` — returns reversed iterator over keys of d:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> for n in reversed(d):
...     print(n)
...
Tina
Bob
Joan
```

- `d.update(arg)` — if arg is itself a mapping, items of arg are added to d overwriting values for keys existing also in d; if arg is an iterable, its elements must be themselves iterables yielding exactly two values, first of which will be a key and the second the corresponding value:

```
>>> d = dict(Joan='Turner', Bob='Cocker')
>>> d1 = {'Joan': 'Baez'}
>>> d.update(d1)
>>> d
{'Joan': 'Baez', 'Bob': 'Cocker'}
>>> ls = [('Bob', 'Dylan'), ['Joe', 'Cocker']]
>>> d.update(ls)
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan', 'Joe': 'Cocker'}
```

- `d | dic` — returns a new dictionary merging items from d with those from dic (which must itself be a dictionary); values from dic overwrite those from d if keys are the same:

```
>>> d1 = dict(Joan='Turner', Bob='Cocker')
>>> d2 = dict(Bob='Dylan', Joe='Cocker', Joan='Baez')
>>> d3 = d1 | d2
>>> d3
{'Joan': 'Baez', 'Bob': 'Dylan', 'Joe': 'Cocker'}
```

[1] Remember that dictionaries always “remember” the order the items were inserted to them.

● `d.keys()`, `d.values()`, `d.items()` — returns a *view* of all keys, values, or items of `d`, respectively. View objects provide a *dynamic view* of the dictionary's entries. No copies are made. *Dynamic* means, that when the dictionary is modified, its views reflect the modifications. Views are themselves iterable and support membership tests. For example:

```
d = dict(Joan='Baez', Bob='Cocker', Tina='Turner')
v = d.items()
print(type(v), '\n')

for it in v:
    print(it)

print()

d.update(Bob='Dylan')
for it in v:
    print(it)

print()

print(('Joan', 'Baez')  in v)
print(('Joe', 'Cocker') in v)
```

prints

```
<class 'dict_items'>

('Joan', 'Baez')
('Bob', 'Cocker')
('Tina', 'Turner')

('Joan', 'Baez')
('Bob', 'Dylan')
('Tina', 'Turner')

True
False
```

Dictionaries can be compared for equality (but not inequalities like <= or >). To be equal, two dictionaries have to contain the same key/value pairs, but not necessarily in the same order:

```
>>> d1 = {'Bob': 'Dylan', 'Joan': 'Baez'}
>>> d2 = {'Joan': 'Baez', 'Bob': 'Dylan'}
>>> d1 == d2
True
```

7.5 Other collections

Besides the collections that are built-in into the language, there are additional collections available in the standard library. We will mention few of them.

7.5.1 Factory function *namedtuple*

The **namedtuple** function (from the ***collections*** module) returns a new *type* which is a subclass of the "normal" **tuple**. The class adds some useful features to "bare" tuples. Normally, we can access element of a tuple by indices; in named tuples, number of elements is fixed and elements on any given position have a name (as names of members in a C **struct**ure). Of course, indices and iterations can still be used, if desired, because names tuples *are* tuples.

Moreover, the types returned by **namedtuple** have helpful docstrings and **__repr__** method already implemented.

The function resides in the standard ***collections*** module, and can be called like this:

```
collections.namedtuple(typename, field_names, *,
                       rename=False, defaults=None, module=None)
```

Its first argument, typename, is the required name of the new type. The result, which is a type, should be stored in a variable with the same name, although it's formally not required. After the name, we pass names of elements on consecutive positions: either as a sequence of strings (e.g.,`['red', 'green', 'blue']`), or as a single string with names separated by white spaces and/or comma (e.g., `'r g b'` or `'r, g, b'`). There are also some optional keyword arguments, but they are seldom used.

When we have our type, we can create objects of this type by passing to the constructor appropriate values — we can use positional arguments, or keyword arguments; in the latter case, order is irrelevant and you can just unpack an existing dictionary (as shown bellow). Named tuples have also a class method **_make(it)** which returns an object given it, where it is an iterable yielding values for all the fields.

Fields of a named tuple can be accessed either by index or by using dot notation (as fields of a C-structure), or by using the built-in function **getattr**. As named tuples are tuples, we can also unpack their elements to named variables (see sec. 7.6, p. 153).

```
>>> import collections
>>> Color = collections.namedtuple('Color', 'r g b')
>>>                                  # type     fields
>>>
>>> orange = Color(255, 165, 0)       # positional arguments
>>> orange[0], orange[1], orange[2]   # by index
(255, 165, 0)
>>> orange.r, orange.g, orange.b      # by name
(255, 165, 0)
>>> for e in orange:                  # iteration
...     print(e)
...
255
165
0
>>> peach = Color(b=180, g=229, r=255) # keyword arguments
>>> peach
Color(r=255, g=229, b=180)             # __repr__ defined
>>>
>>> red, green, blue = peach           # unpacking
>>> red, green, blue
(255, 229, 180)
```

```
>>>
>>> it = [255, 127, 80]
>>> coral = Color._make(it)            # from iterable
>>> coral
Color(r=255, g=127, b=80)
>>> getattr(coral, 'g')                # accessing field
127
>>>
>>> d = {'r': 93, 'g': 63, 'b': 211}
>>> iris = Color(**d)                  # unpacking dict
>>> iris
Color(r=93, g=63, b=211)
```

Named tuples, being tuples, have all methods of regular tuples, but also three additional methods and two fields. One of the methods, **_make**, was already mentioned; the two remaining are (below nt stands for a named tuple):

- nt._asdict() — returns a dictionary representing nt but as a dictionary with fields as keys:

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p._asdict()
{'x': 2, 'y': 5}
```

- nt._replace(**kwargs) — returns a new named tuple with some or all fields replaced by values specified by kwargs:

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p1 = p._replace(y=4)
>>> p2 = p._replace(x=3, y=6)
>>> p1, p2
(Point(x=2, y=4), Point(x=3, y=6))
```

- nt._fields — returns the attribute _fields which is a tuple of names of fields:

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p._fields
('x', 'y')
>>> Point._fields   # this is class field
('x', 'y')
```

This attribute may be useful when creating new named-tuple types “extending”, in some sense, existing ones:

```
>>> Pixel = namedtuple('Pixel', ('shade',)+Point._fields)
>>> px = Pixel(255, 2, 5)
>>> px
Pixel(shade=255, x=2, y=5)
```

7.5.2 Class *defaultdict*

The **defaultdict** class (from the ***collections*** module) inherits from the built-in **dict** class and hence supports all of its operations. There is one difference though: we can provide a way to handle the situation, when an access to a dictionary with a non-existing key is made. Thus, instead of checking for the existence of a key and then adding an entry if it's missing, we can just declare the default. As we will see, this is really useful and powerful feature of the **defaultdict** class.

The **defaultdict** constructor can be invoked like this:

```
collections.defaultdict(callableOb=None, /[, ...])
```

The first argument, if not **None** which is the default, must be a callable object. The remaining arguments are exactly the same as for the "regular" **dict** class (see sec. 7.4, p. 137).

If callableOb is missing or is **None**, the object returned by the constructor behaves as "normal" **dict** object; in particular, if we try to access an item using a non-existing key, a **KeyError** is raised.

What happens if callableOb exists? Then, when we try to use non-existing key, this callable object is called with no arguments and a new item is added to the dictionary: with this key and whatever was returned by the invocation of callableOb as the associated value.

Anything that can be invoked without arguments can play the rôle of callableOb. Most often, however, this is just the name of a class: as we remember object representing classes are callable, and without arguments return the default value of a given type (zero for numeric types, empty collection for collection types, etc.)

An example. We have a list of pairs (2-tuples) of cities, where *(city1, city2)* means that there is a direct flight from *city1* to *city2*. What we want is a dictionary with cities as keys and sets of possible destinations from these cities as values. That's just one loop with one-line body, with no **if**s!

Listing 36 ACC-DefDict/DefDict.py

```python
#!/usr/bin/env python

from collections import defaultdict

flights = [ ('Athens', 'Basel'),       ('Athens', 'Copenhagen'),
            ('Basel', 'Athens'),       ('Basel', 'Delhi'),
            ('Copenhagen', 'Athens'), ('Delhi', 'Athens'),
            ('Delhi', 'Basel'),        ('Delhi', 'Copenhagen') ]

destinations = defaultdict(set)  ❶

for start, to in flights:
    destinations[start].add(to)  ❷

  # just printing
for start, dests in destinations.items():
    print(f'{start:10} -> {str(dests)}')
```

In line ❶, we create a defaultdict with set as the callable object. Now, in line ❷,

- if the city represented by start appears for the first time, new item is created with start as the key and an empty set as the value, to which immediately to is added;
- if the city represented by start already exists as a key in the dictionary, then to is just added to the already existing set of destinations available from this city.

The program prints

```
Athens     -> {'Basel', 'Copenhagen'}
Basel      -> {'Delhi', 'Athens'}
Copenhagen -> {'Athens'}
Delhi      -> {'Basel', 'Athens', 'Copenhagen'}
```

Another example: This time with int as the callableOb. We have list of purchases with prices and we want to calculate total amount of money spent on different products:

Listing 37 ACE-Purchases/Purchases.py

```python
#!/usr/bin/env python

from collections import defaultdict

purchases = [ ('Carrot', 3),   ('Beetroot', 7),
              ('Apples', 9),   ('Beetroot', 8),
              ('Apples', 8),   ('Apples', 10),
              ('Beetroot', 6), ('Carrot', 4) ]

totals = defaultdict(int)

for prod, price in purchases:
    totals[prod] += price      ❶

  # just printing
for prod, tot in totals.items():
    print(f'{prod:8} -> {tot:2}')
```

and the output is

```
Carrot   ->  7
Beetroot -> 21
Apples   -> 27
```

It's again a loop with just one-line body (❶)!

7.5.3 Class *Counter*

The Counter class (from the *collections* module) is a subclass of dict designed specifically for keeping track of the number of occurrences of its elements. We can say that it's a collection where elements are stored as keys (and therefore must be hashable!) and their counts (multiplicities) as values.[1] It seems to follow that the values should always be positive integer numbers, but in fact zero and negative values *are* allowed. Moreover, their type doesn't need to be **int** — fractions, floats, and decimals will also work, as generally any type supporting addition, subtraction and comparisons. However, in most cases, the values are just integer numbers.

The **Counter** constructor can be invoked like this:

```
collections.Counter([iterable-or-mapping])
```

If the constructor's argument is an iterable, occurrences are counted:

```
>>> import collections
>>> collections.Counter('abracadabra')
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
```

As we can see, elements are printed in decreasing order of values. However, when iterated, counters behave like dictionaries — the order in which elements were first encountered:

```
>>> import collections
>>> c = collections.Counter([1, 5, 4, 2, 4, 4, 1, 2, 3, 3])
>>> c
Counter({4: 3, 1: 2, 2: 2, 3: 2, 5: 1})
>>> for p in c.items():
...     p
...
(1, 2)
(5, 1)
(4, 3)
(2, 2)
(3, 2)
```

We can also pass a mapping to the constructor (as **dict**, **defaultdict**, **Counter**, or as keyword arguments):

```
>>> from collections import Counter
>>> Counter()                                   # empty
Counter()
>>> Counter('azazello')                         # sequence
Counter({'a': 2, 'z': 2, 'l': 2, 'e': 1, 'o': 1})
>>> Counter({'a': 3, 'b': 2})                   # mapping
Counter({'a': 3, 'b': 2})
>>> Counter(a=3, b=2)                           # keyword args.
Counter({'a': 3, 'b': 2})
```

An attempt to access an element with non-existing key does not raise any exception but returns 0; if it is an assignment, a new element is added with this key and the value 0 (which we can immediately modify):

```
>>> from collections import Counter
>>> c = Counter()
```

[1] Similarly to bags or multisets in other languages; these are sets, but in which elements can occur more than once.

```
>>> for letter in 'arrabal':
...     c[letter] += 1  # item with key letter added
...
>>> c
Counter({'a': 3, 'r': 2, 'b': 1, 'l': 1})
>>> c['z'] # no 'z', no assignment; 0 returned without adding
0
>>> c
Counter({'a': 3, 'r': 2, 'b': 1, 'l': 1})
```

Counters are mappings, so they inherit methods from **dict**. However, implementation of class method **fromkeys** is removed, and implementation of **update** is redefined: when an existing item is updated, the new value does not replace the old one, but rather is added to it:

```
>>> from collections import Counter
>>> c = Counter({'a': 3, 'b': 2, 'c': 1})
>>> d = dict(a=2, b=-2, c=-2, d=3)
>>> c.update(d)
>>> c
Counter({'a': 5, 'd': 3, 'b': 0, 'c': -1})
```

There are two additional methods of class **Counter**, not inherited from **dict** (cn below denotes a counter):

- `cn.elements()` — returns an iterator over elements (keys, not items) each of then will appear as many times as its count; elements with non-positive will be ignored:

```
>>> from collections import Counter
>>> c = Counter({'a': 2, 'b': 0, 'c': -1, 'd': 3})
>>> for e in c.elements():
...     e
...
'a'
'a'
'd'
'd'
'd'
```

- `cn.most_common([n])` — returns a list of the n most common (with highest counts) elements and their counts, ordered by counts in the decreasing order. If n is omitted or **None**, the methods returns all elements. As it returns a list (of 2-tuples), one can easily use slicing to get the n *least* common elements (for example, by counts in the increasing order, as below):

```
>>> from collections import Counter
>>> c = Counter({'a': 6, 'b': 10, 'c': 1, 'd': -2})
>>> n = 2
>>> c.most_common(n)
[('b', 10), ('a', 6)]
>>> c.most_common()[:-n-1:-1]
[('d', -2), ('c', 1)]
```

- `cn.subtract([iterable-or-mapping])` — as (update), but subtracts counts instead of adding them (values missing in `cn` are assumed to be 0):

```
>>> from collections import Counter
>>> c1 = Counter({'a': 6, 'b': 10, 'c': 1})
>>> c2 = Counter({'a': 4, 'b': 10, 'd': 2})
>>> c1.subtract(c2)
>>> c1
Counter({'a': 2, 'c': 1, 'b': 0, 'd': -2})
```

- `cn.total()` — returns the sum of all counts:

```
>>> from collections import Counter
>>> c = Counter({'a': 6, 'b': 10, 'c': -2})
>>> c.total()
14
```

We can use rich comparison operators (==, !=, <, <=, >, >=) for counters, as we could for "normal" sets (see sec. 7.3, p. 130. Elements which are present in a counter on one side of these operators, but are missing on the other side, are assumed to exist and have count zero, so

```
>>> Counter(b=1) == Counter(a=0, b=1, c=0)
True
```

What is compared, are really counts, so

```
>>> Counter(a=-1, b=1) < Counter(b=2, c=1)
True
```

as on the left missing 'c' is assumed to have count 0, and similarly 'a' on the right.

Also, set calculus operators can be used, as for sets, but now everything works on counts of elements, again assuming that missing elements have count 0. Both counters-operands, on the left and on the right hand side of operators, can contain elements with counts zero or less. However, the resulting counter will be a true multiset: all elements with counts less than 1 will be removed.

Addition and subtraction just add or subtract counts (and remove elements with non-positive counts in the result):

```
>>> Counter(a=-1, b=1) + Counter(a=2, b=2, c=1, d=-1)
Counter({'b': 3, 'a': 1, 'c': 1})
>>> Counter(a=2, b=2, c=1, d=-1) - Counter(a=3, b=-3, c=-1)
Counter({'b': 5, 'c': 2})
```

Intersection and union calculate the minimum and maximum of corresponding counts:

```
>>> Counter(a=2, b=2, c=1, d=-1) & Counter(a=3, b=-3, c=-1)
Counter({'a': 2})
>>> Counter(a=2, b=2, c=1, d=-1) | Counter(a=3, b=-3, c=-1)
Counter({'a': 3, 'b': 2, 'c': 1})
```

Finally, there are unary + and − operators: the first is equivalent to adding (with +) the empty counter, what boils down to removing element with non-positive counts

```
>>> Counter(a=2, b=2, c=0, d=-1) + Counter()
Counter({'a': 2, 'b': 2})
>>> +Counter(a=2, b=2, c=0, d=-1)
Counter({'a': 2, 'b': 2})
```

and the second is equivalent to subtracting (with $-$) a given counter *from* the empty counter, i.e., flipping the sign of all counts and then leaving only those which are now positive:

```
>>> Counter() - Counter(a=2, b=-2, c=0, d=-1)
Counter({'b': 2, 'd': 1})
>>> -Counter(a=2, b=-2, c=0, d=-1)
Counter({'b': 2, 'd': 1})
```

7.5.4 Class *deque*

Objects of class **deque** represent deques:[1] sequences with fast (complexity $\mathcal{O}(1)$) operations on both ends, but possibly $\mathcal{O}(n)$ for elements somewhere inside. In Python, deques are implemented as doubly linked lists. They can provide a simple implementations of stacks or lists. Normally, to implement them, doubly linked list would be an overkill, but is nonetheless acceptable in Python, because deques are fully implemented in C (at least in standard CPython), so their performance is quite satisfactory.

Deques are created by using the constructor

```
collections.deque([iterable[, maxlen]])
```

which returns a deque object initialized with data from iterable (empty if iterable is not specified). The optional second argument, maxlen, sets the maximum number of elements in the created deque. Then, if the deque is full and new elements are added at the beginning or at the end, the corresponding number of elements is removed on the opposite side. If maxlen is not specified (or is **None**), the size is unbounded. The value of maxlen is available as a field maxlen of the deque (**None** for unbounded deques).

Deques, being sequences, support iteration, functions **len(d)** and **reversed(d)**, copying, membership testing with **in** and **not in**, and also indexing and slicing, although it has to be kept in mind that indexing is generally an $\mathcal{O}(n)$ operation for deques.

Methods supported by deques are shown below (d denotes a deque):

● `d.append(e)`, `d.appendleft(e)` — adds element e at the end (beginning) of d. If the deck is bounded and full, the first element on the other side is removed:

```
>>> from collections import deque
>>> d = deque([], 3)  # empty and bounded
>>> d
deque([], maxlen=3)
>>> d.append(2)
>>> d.appendleft(1)
>>> d.append(3)
>>> d.append(4)       # fourth element?
>>> d
deque([2, 3, 4], maxlen=3)
```

[1] Pronounced as ‘deck’; from *double-ended queue*.

- `d.insert(ind, e)` — inserts e at position (index) ind. Raises IndexError if it would cause the length of d to exceed maxlen:

```
>>> from collections import deque
>>> d = deque([1, 2, 5, 6], 5)
>>> d.insert(2, 3)
>>> d
deque([1, 2, 3, 5, 6], maxlen=5)
>>> d.insert(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: deque already at its maximum size
```

- `d.extend(iterable)`, `d.extendleft(iterable)` — adds elements from iterable at the end (beginning) of d. If the deck is bounded, some elements from the other side may be removed, for the length not to exceed maxlen:

```
>>> from collections import deque
>>> d = deque([3, 4], 6)     # bounded
>>> d.extendleft((2, 1))     # note the order
>>> d
deque([1, 2, 3, 4], maxlen=6)
>>> d.extend([5, 6, 7, 8])
>>> d
deque([3, 4, 5, 6, 7, 8], maxlen=6)
```

- `d.pop()`, `d.popleft()` — returns the last (first) element form d and removes it:

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 4, 5])
>>> d.pop()
5
>>> d.pop()
4
>>> d.popleft()
1
>>> d
deque([2, 3])
```

- `d.remove(val)` — removes the first element equal to val, or raises ValueError if there is no such element in d:

```
>>> from collections import deque
>>> d = deque([4, 3, 5, 2, 1])
>>> d.remove(5)
>>> d
deque([4, 3, 2, 1])
>>> d.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in deque
```

- `d.reverse()` — reverses in place the order of elements in d:

```
>>> from collections import deque
>>> d = deque([4, 3, 2, 1])
>>> d
deque([4, 3, 2, 1])
>>> d.reverse()
>>> d
deque([1, 2, 3, 4])
```

- `d.index(val[, start[, stop]])` — returns the index of the first element equal to val, optionally taking into account only subsequence on positions from start to stop-1; raises **ValueError** if there is no such element. Defaults for start and stop are 0 and len(d), respectively:

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 3, 1, 4])
>>> d.index(2)
1
>>> d.index(1, 1)
4
>>> d.index(1, 0, 4)
0
>>> d.index(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in deque
```

- `d.clear()` — removes all elements from d leaving it empty.

- `d.copy()` — returns a *shallow* copy of d:

```
>>> from collections import deque
>>> d = deque([ (1, 2), [3, 4] ])
>>> dc = d.copy()
>>> d
deque([(1, 2), [3, 4]])
>>> dc
deque([(1, 2), [3, 4]])
>>> d[1][1] = 'X'
>>> d
deque([(1, 2), [3, 'X']])
>>> dc
deque([(1, 2), [3, 'X']])
```

- `d.count(val)` — returns number of elements equal to val.

- `d.rotate(n=1)` — rotates elements of the deque by n positions to the right (to the left if n is negative):

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 4, 5, 6, 7])
>>> d.rotate(2)
>>> d
deque([6, 7, 1, 2, 3, 4, 5])
>>> d.rotate(-4)
>>> d
deque([3, 4, 5, 6, 7, 1, 2])
>>> d.rotate()
>>> d
deque([2, 3, 4, 5, 6, 7, 1])
```

Bounded deques can be used in a very common situation when we want to remember last n values form a sequence of data, like last n telephone calls, last n data-base queries, etc.:

```
>>> from collections import deque
>>>
>>> calls = ['Mike', 'Jim', 'Alice', 'Kate', 'Bob', 'Sue']
>>> lastcalls = deque([], maxlen=3)
>>> for call in calls:
...     lastcalls.append(call)
...     lastcalls
...
deque(['Mike'], maxlen=3)
deque(['Mike', 'Jim'], maxlen=3)
deque(['Mike', 'Jim', 'Alice'], maxlen=3)
deque(['Jim', 'Alice', 'Kate'], maxlen=3)
deque(['Alice', 'Kate', 'Bob'], maxlen=3)
deque(['Kate', 'Bob', 'Sue'], maxlen=3)
```

7.6 Unpacking iterables

Iterable objects (sufficiently "small"...) can be easily "unpacked" to a tuple or list of individual variables with different names. This is a kind of assignment: on the left-hand side there is a tuple (usually without parentheses) or list of variables/names to which elements of an iterable on the right-hand side are to be assigned. The number of these elements must be the same as the number of variables on the left.
What's on the right may be anything, as long as it's iterable: it may be a string, a list, a tuple, a slice, a **range** object, a set, a dictionary, a generator:

```
  # 1. to tuple
a, b, c, d = range(4)
print('** 1 ** ', f'{a=}, {b=}, {c=}, {d=}')

  # 2. to list
[e, f, g, h] = range(4)
print('** 2 ** ', f'{e=}, {f=}, {g=}, {h=}')

  # 3. from slice
ls = [0, 1, 2, 'three', 4, 5, 6, 'seven', 8, 'nine']
```

```
i, j, k, l = ls[1:8:2]
print('** 3 ** ', f'{i=}, {j=}, {k=}, {l=}')

  # 4. elements can be tuples or lists themselves
m, n, o, p = (1, 2, (3, 4, 5), [6, 7])
print('** 4 ** ', f'{m=}, {n=}, {o=}, {p=}')

  # 5. from string
q, r, s = 'QRS'
print('** 5 ** ', f'{q=}, {r=}, {s=}')

  # 6. from slice of string
t, u, v = '012345678'[2:9:3]
print('** 6 ** ', f'{t=}, {u=}, {v=}')

  # 7. from generator
w, x, y = (z**3 for z in range(1,10) if z%3 == 1)
print('** 7 ** ', f'{w=}, {x=}, {y=}')

  # 8. number of elements mismatch -> exception
p, q, r = range(4)
```

The program prints

```
** 1 **  a=0, b=1, c=2, d=3
** 2 **  e=0, f=1, g=2, h=3
** 3 **  i=1, j='three', k=5, l='seven'
** 4 **  m=1, n=2, o=(3, 4, 5), p=[6, 7]
** 5 **  q='Q', r='R', s='S'
** 6 **  t='2', u='5', v='8'
** 7 **  w=1, x=64, y=343
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578,
    in runcode exec(code, self.locals)
  File "/home/werner/p.py", line 32, in <module>
    p, q, r = range(4)
ValueError: too many values to unpack (expected 3)
```

Sets are iterable as well, but there might be a problem with them, as we cannot predict the order in which elements will be assigned to variables on the left-hand side:

```
a, b, c, d = {1, 'Ann', 'Barbra', (2, 'c')}
print(f'{a=}, {b=}, {c=}, {d=}')
```

printed

```
a='Ann', b=(2, 'c'), c='Barbra', d=1
```

but there is no guarantee that the order will always be the same.

Iteration over a dictionary yields its keys. We can, however, iterate over the key-value pairs (2-tuples) by calling **items** on a dictionary. These tuples are themselves iterable, so we can unpack them to separate variables, as shown below in the third example of the program

```
dc = dict(x='Ann', y=23, z=[1, 2])

  # 1. keys
a, b, c = dc
print('*1*', f'{a=}, {b=}, {c=}')

  # 2. items (key-value tuples)
d, e, f = dc.items()
print('*2*', f'{d=}, {e=}, {f=}')

  # 3. nested unpacking
(k1, v1), kv, (k3, v3) = dc.items()
print('*3*', f'{k1=}, {v1=}, {kv=}, {k3=}, {v3=}')
```

which prints

```
*1* a='x', b='y', c='z'
*2* d=('x', 'Ann'), e=('y', 23), f=('z', [1, 2])
*3* k1='x', v1='Ann', kv=('y', 23), k3='z', v3=[1, 2]
```

Unpacking is very often used in loops, when elements that we get are iterable (as entries of a dictionary, or tuples). Here also we can use nested unpacking; for example the snippet fragment

```
triangle = ( ("A", (1, 5)), ("B", (5, 3)), ("C", (2, 1)))

for symb, (x, y) in triangle:
    print(f'{symb} = ({x=}, {y=})')
```

prints

```
A = (x=1, y=5)
B = (x=5, y=3)
C = (x=2, y=1)
```

The name of one, and at most one, variable on the left may be prepended by a star (e.g., `*v`), which denotes the **unpacking operator**. It means that elements which have not been paired with non-starred variables appearing before and after `*v` will be “collected”, in the form of a list, in the starred variable `v`. For example, the program

```
tp = (1, 'two', [3, 4], {5, 'six'})

u, x, y, z = tp
print('*1*', f'{u=}, {x=}, {y=}, {z=}')

u, x, *y = tp
print('*2*', f'{u=}, {x=}, {y=}')

u, *x, z = tp
print('*3*', f'{u=}, {x=}, {z=}')

u, *_, z = tp  # convention: only u and z are of interest
print('*4*', f'{u=}, {z=}')
```

```
*u, y, z = tp
print('*5*', f'{u=}, {y=}, {z=}')
```

prints

```
*1* u=1, x='two', y=[3, 4], z={5, 'six'}
*2* u=1, x='two', y=[[3, 4], {5, 'six'}]
*3* u=1, x=['two', [3, 4]], z={5, 'six'}
*4* u=1, z={5, 'six'}
*5* u=[1, 'two'], y=[3, 4], z={5, 'six'}
```

The list corresponding to the starred variable is very often not needed; what we want to "extract" are those values which are assigned to the non-starred variables. To emphasize this fact, according to a popular convention, the starred variable is given the name _ (underscore), as in the 4th example in the program above; this is a valid name of a variable, but by convention it suggests that its appearance here is forced by the syntax — its value is irrelevant and will not be used.

Unpacking tuples or lists is very often used for swapping values of variables:

```
>>> a, b = 1, 2
>>> a, b
(1, 2)
>>> b, a = a, b     # swap
>>> a, b
(2, 1)
>>>
>>> a, b, c, d = 'three', 'four', 'one', 'two'
>>> a, b, c, d
('three', 'four', 'one', 'two')
>>> a, b, c, d = c, d, a, b
>>> a, b, c, d
('one', 'two', 'three', 'four')
```

Many functions return their results in the form of tuples (or generators, as in the example below). It is then very common to unpack the result to named variables (those not interesting for us to _), as in the following program

```
def powers1_5(n):
    v = 1
    for i in range(1, 6):
        yield (v := v*n)

a, b, c, d, e = powers1_5(3)
print(a, b, c, d, e)

_, square, cube, *_ = powers1_5(3)
print(f'{square=}, {cube=}')
```

which prints

```
3 9 27 81 243
square=9, cube=27
```

Unpacking starred variables can also simplify merging several sequences (more generally: iterables) to one, as in the following fragment

```
st = {1, 2}
ls = [3, 4, 5]
def seq(n, m):
    for i in range(n, m+1):
        yield i

r = [*st, *ls, *seq(6, 7)]
print(r)
```

which prints

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

Dictionaries can be unpacked by a single star, and then we get just its keys, or by double star to get key-value tuples:

```
d1 = dict(a=1, b=2, c=3)
d2 = dict(c='three', d=4)

dk = {*d1, *d2}   # unpacking keys, dk is set
print(dk)

di = {**d1, **d2} # unpacking items, di is dict
print(di)

def getdic():
    return { 'e': 5, 'f': 6}

dd = {**di, **getdic()}  # dd is dict
print(dd)
```

prints

```
{'a', 'c', 'd', 'b'}
{'a': 1, 'b': 2, 'c': 'three', 'd': 4}
{'a': 1, 'b': 2, 'c': 'three', 'd': 4, 'e': 5, 'f': 6}
```

Chapter 8

Strings

String (objects of class **str**) are a very special type of *immutable* sequential collections. Due to their significance and specific interpretation, they have many methods not present in other sequences.

String are sequences of characters, although there is no separate type **char** in Python — object of this type known from other languages correspond to just one-character strings. By definition, string are internally represented in the UTF-8 encoding, so one character can occupy from one to four bytes.[1]

String literals may be enclosed in single quotes (’like this’) or in double quotes ("like this"). In the first case, double quotes may be embedded without escaping them, and the other way around in the second case. Two string literals adjacent to each other with only white space between them are converted to a single string (’This ’ "and"" that" is equivalent to a single string ’This and that’). There is also the triple-quoted form (”’like this”’ or """this"""); all whitespace characters, including new-line characters, will be then included in the string literal:

```
s = '''All white-space
characters
    included'''
print(s)
```

prints

```
All white-space
characters
    included
```

Inside literal strings some ‘special’ characters must be escaped with the \ character, similarly to what we also have in C/C++/Java:

Table 3: Escape sequences

SEQUENCE	DESCRIPTION
\<newline>	ignored
\\	backslash
\’	single quote
\"	double quote
\a	BEL (alarm)
\b	BS (backspace)
\f	FF (form feed)
\n	LF (line feed, new line)
\r	CR (carriage return)
\t	TAB (horizontal Tab)
\v	VT (vertical Tab)
\ooo	character with octal value *ooo*
\hh	8-bit character with hex value *hh*

[1] https://en.wikipedia.org/wiki/UTF-8

Table 3: (continued)

SEQUENCE	DESCRIPTION
\uhhhh	16-bit character with hex value *hhhh*
\Uhhhhhhhh	32-bit character with hex value *hhhhhhhh*

However, for the so called 'raw' strings (prefixed with the letter 'r' or 'R'), backslash stands just for itself — this is useful when defining regular expressions

```
>>> ss =  '\a\b\r\n'
>>> sr = r'\a\b\r\n'
>>> len(ss), len(sr)
(4, 8)
>>> sr
'\\a\\b\\r\\n'
```

One-parameter constructor of the **str** class accepting an object of any type corresponds roughly to the **toString** method from Java — it returns a string describing the object. It does so by invoking the **__str__** method on the object passed as the argument, and if there is no this method, **__repr__** will be called. If both are missing, the default implementation of **__str__** from class **object** will be used:

```
class A:                        # only str
    def __str__(self):
        return "A(__str__)"

class B:                        # only repr
    def __repr__(self):
        return "B(__repr__)"

class C:                        # both
    def __str__(self):
        return "C(__str__)"
    def __repr__(self):
        return "C(__repr__)"

class D:                        # neither
    pass

print(str(A()), str(B()), str(C()))
print(str(D()))
```

prints

```
A(__str__) B(__repr__) C(__str__)
<__main__.D object at 0x7f8f1a63bd60>
```

If a byte object (like **bytes** or **bytearray**) is the first parameter and at least one more parameter has been provided (specifying an encoding), then the behavior is slightly different (see the next sections).

Out of many methods of the str class,[1] we will mention some, probably most often used:

Checking type of characters in strings

Several methods check the type of characters of strings. They all return True or False. Their names are similar to what we know from standard C[2]; there are also analogous static methods of the Character class in Java. There is a difference, though: in Python they are nonstatic methods of the str class and operate on *strings*, not on single characters. For example

```
>>> '+123'.isdigit()
False
>>> '123'.isdigit()
True
```

because in the first case, the string contains '+' character, which is not a digit. On the other hand, methods like isupper or islower check only characters for which cases make sens (*cased* characters), so

```
>>> 'U.S.A.'.isupper()
True
>>> 'England'.isupper()
False
```

The following methods belong to this group of methods of the str class: isalnum(), isalpha(), isascii(), isdecimal(), isdigit(), isidentifier(), keyword.iskeyword(), islower(), isnumeric(), isprintable(), isspace(), istitle(), isupper().

Transforming strings

Another group of methods returns a string which is a transformation of the string on which these methods are invoked. We can mention here (s is the string we invoke a given method on):

s.capitalize(), s.title() — capitalize returns a string with the first letter capitalized, if it is a lowercase letter; all other cased characters are lowercased, while not-cased characters are left intact. The title method capitalizes all words separately, where by 'word' a sequence of letters is meant:

```
>>> 'aDaM 12, BARBRA 18'.capitalize()
'Adam 12, barbra 18'
>>> '12 aDaM, 18 BARBRA'.capitalize()
'12 adam, 18 barbra'
>>> '12 aDaM, 18 BARBRA'.title()
'12 Adam, 18 Barbra'
```

s.ljust(width[, fillchar]), s.rjust(width[, fillchar]), s.center(width[, fillchar]) — returns a string of length width containing s at the beginning (ljust), at the end (rjust)) or in the middle (center) and padded after, before, or at both sides of s, respectively, with character fillchar (by default a space) repeated as many times as needed to get a string of length width; if width is not greater then len(s), s is returned:

[1] https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

[2] From the ***ctype.h*** header in C and ***cctype*** in C++.

```
>>> 'Adam'.ljust(10)
'Adam      '
>>> 'Adam'.ljust(10, '-')
'Adam------'
>>> 'Adam'.rjust(10, '=')
'======Adam'
>>> 'Adam'.center(10, '.')
'...Adam...'
```

s.zfill(width) — similar to **rjust**, but characters ’0’ (zero) are added at the beginning, and if the *first* character of s is ’−’ (minus), the first zero is replaced by ’−’; if width is not greater than len(s), s is returned. The method is useful mainly for strings representing numbers:

```
>>> '123'.zfill(6)
'000123'
>>> '-123'.zfill(6)
'-00123'
>>> ' -123'.zfill(6)    # minus is not the first character
'0 -123'
>>> '-abc'.zfill(6)
'-00abc'
```

s.lstrip([chars]), **s.rstrip([chars])**, **s.strip([chars])** — returns the string without the prefix (for **lstrip**), or suffix (for **rstrip**), or both suffix and prefix (for **strip**) consisting *only* of characters present in [chars]. If [chars] is not specified or is None, whitespace characters are “removed”:

```
>>> '11 23 Adam 99 34'.lstrip('0123456789 ')
'Adam 99 34'
>>> '11 23 Adam 99 34'.rstrip('0123456789 ')
'11 23 Adam'
>>> '    Adam\t\r\n'.strip()
'Adam'
```

s.removeprefix(prefix), **s.removesuffix(suffix)** — returns the string without the prefix (for **removeprefix**) or suffix (for **removesuffix**) if it is exactly the same as the given argument; if not, s is returned:

```
>>> 'Mr Jones'.removeprefix('Mr ')
'Jones'
>>> 'Mona Lisa.jpg'.removesuffix('.jpg')
'Mona Lisa'
```

s.lower(), **s.upper()**, **s.swapcase()** — returns the string with all upper-case letters replaced by the corresponding lower-case letters (**lower**), the other way around (**upper**) or both (**swapcase**):

```
>>> 'bArBrA 18'.lower()
'barbra 18'
```

```
>>> 'bArBrA 18'.upper()
'BARBRA 18'
>>> 'bArBrA 18'.swapcase()
'BaRbRa 18'
```

s.replace(old, new[, count]) — returns a copy of s with all occurrences of old replaced by new. If count is given, only the first count occurrences are replaced.

```
>>> 'AA BB AAA C AAAA D AAAAA'.replace('AA', 'X')
'X BB XA C XX D XXA'
>>> 'AA BB AAA C AAAA D AAAAA'.replace('AA', 'X', 3)
'X BB XA C XAA D AAAAA'
```

Finding substrings

A few methods of the **str** class are related to searching for substrings in a given string. The following methods fall into this category.

s.find(sub[, start[, end]]), **s.rfind(sub[, start[, end]])**,
s.index(sub[, start[, end]]), **s.index(sub[, start[, end]])** — returns the first (**find** and **index**) or the last (**rfind** and **rindex**) index in s where substring sub is found. If optional arguments are given, sub is searched for in the part s[start:end] only; start defaults to 0, end to len(s). If sub doesn't occur in s, -1 is returned for **find** and **rfind**, while **ValueError** exception is raised for **index** and **rindex**:

```
>>> s = 'barbaric'
>>> s.find('bar'), s.rfind('bar')
(0, 3)
>>> s.index('bar'), s.rindex('bar')
(0, 3)
>>> s.find('br')
-1
>>> s.index('br')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

s.count(sub[, start[, end]]) — returns the number of non-overlapping occurrences of substring sub in s, or in the slice s[start:end] if optional arguments are given:

```
>>> 'bcbcbcb'.count('bcb')
2
```

Note that substring 'bcb' starting at the second 'b' was not counted, as it partly overlaps with the first 'bcb' at the very beginning of the string.

s.endswith(sub[, start[, end]]), **s.startswith(sub[, start[, end]])** — checks if s, or its slice s[start:end] if optional arguments have been given, has substring sub at the very end or at the very beginning, respectively; returns **True** or **False**. The sub argument is treated literally, *not* as a regular expression.

```
>>> 'caravaggio.jpg'.endswith('.jpg')
True
>>> 'caravaggio.jpg'.startswith('Car')
False
```

Splitting

There are also several methods which split a given string into parts.

s.split(sep=None, maxsplit=-1), **s.rsplit(sep=None, maxsplit=-1)** — returns a list of the "words" in s, treating sep as the separator between them. If maxsplit is given, it is the maximum number of splits, so the maximum number of parts ("words") will be maxsplit+1 and the last element of the list will contain all remaining "words" (**split**). The **rsplit** method works "backwards": at most maxsplit splits will be done — the rightmost ones, and the remaining leftmost "words" will go to the first element of the resulting list. If maxsplit is not specified or is -1, then there is no limit on the number of splits. If sep is given, two adjacent separators are understood to delimit an empty string. If s begins with the separator, the first returned "word" will be empty; similarly, if it ends with the separator, the last "word" will be empty. The separator sep may be a string of any length, not necessarily one-character. If sep is not specified or is None, any non-empty sequence of white characters are treated as a single separator, but leading and trailing white characters are ignored (so the resulting list will *not* contain empty elements).

```
>>> ' a   b c   d '.split()
['a', 'b', 'c', 'd']
>>> ':a::b:'.split(':')
['', 'a', '', 'b', '']
>>> 'a b c d e f'.split(maxsplit=3)
['a', 'b', 'c', 'd e f']
>>> 'a b c d e f'.rsplit(maxsplit=3)
['a b c', 'd', 'e', 'f']
```

s.splitlines(keepends=False) — returns a list of the lines in s. Splitting is done at new-line character (\n), carriage return (\r), vertical tabulator (\v), form feed character (\f) and a few others, rarely used. However, combination CR-LF (\r\n) is treated as one single separator. Separators themselves are not included in the resulting lines, unless keepends is True. If there is a separator at the very end of s, it is ignored (but only one, not a sequence of them). Separator at the beginning is *not* ignored.

```
>>> '\nA\r\nB\rC\nD\n\nE\v\f'.splitlines()
['', 'A', 'B', 'C', 'D', '', 'E', '']
```

s.partition(sep), **s.rpartition(sep)** — returns a 3-tuple containing the part before the first occurrence of the separator sep, separator itself, and the remaining part of s. If sep does not occur in s, returns a 3-tuple containing s itself followed by two empty strings. The **rpartition** is similar, but splits s at the *last* occurrence of sep, and if sep doesn't occur in s, the 3-tuple returned contains two empty strings followed by s itself.

```
>>> 'A:B:C:D'.partition(':')
('A', ':', 'B:C:D')
>>> 'A:B:C:D'.rpartition(':')
('A:B:C', ':', 'D')
>>> 'A:B:C:D'.partition('-')
('A:B:C:D', '', '')
>>> 'A:B:C:D'.rpartition('-')
('', '', 'A:B:C:D')
```

Combining strings

There is no type corresponding to mutable strings in Python, as are **string** in C++ or **StringBuffer**/**StringBuilder** in Java. However, in many situations, it is possible to get similar functionality using the **join** method of class **str**.

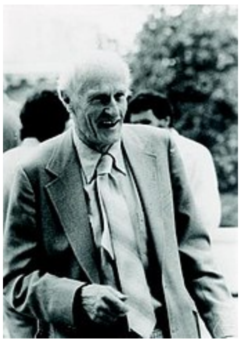
s.join(iterable) — returns a string which is the concatenation of all the strings yielded by iterable. These must be strings only; any non-string value will trigger the **TypeError** exception. The concatenated strings will be separated by s, which may be, and very often is, an empty string.

```
>>> lst = ['To', 'be', 'or', 'not', 'to', 'be']
>>> ' '.join(lst)
'To be or not to be'
>>> '-'.join(a+b for a in ['A', 'B'] for b in ('1', '2'))
'A1-A2-B1-B2'
```

Chapter 9

Regular expressions

Regular expression (**regex**) is a sequence of characters which defines a pattern that we want to search for in a string (generally, in a text which may be arbitrary long). Regexes are 'compiled' into a form resembling functions and executed by the so called *regular expression engines* — almost all modern programming languages support regular expressions (built into the language or as part of their standard libraries). The theory behind regexes is rather involved and its full understanding requires quite advanced mathematical knowledge; it was developed by an outstanding logician Stephen Cole Kleene (pronounced KLAY-nee) in 1950s and first used in practice in early implementations of Unix text processors and utility programs (Ken Thompson). Almost all modern implementations of regular expression engines are based on Larry Wall's implementation in his Perl programming language (late 1980s).



Stephen C. Kleene

For details on the Python implementation: see the documentation[1] (also the tutorial[2] may be useful).

9.1 Basic concepts

Suppose we are looking for a word, say `elephant`, in a text. Then the regular expression which defines this word will be just `"elephant"`. But what if we have, in our text, the word `Elephant`? Of course, this won't match, because the first letter differs. Or, we look for `cat` but only if it is a separate word, not part of another word (like in `tomcat` or `caterpillar`). Or, we are looking for numbers (sequences of digits) but we don't know in advance what numbers occur in our text and of what length (number of digits) they are. All these problems can be easily solved with the help of regular expressions which allow us to formulate such requirements as 'a sequence of letters', 'any uppercase letter followed by a dot', 'a sequence of at least four but at most seven digits the first of which is not 0', 'two words separated by one or more spaces or TAB characters', etc.

Tools for handling regular expression are part of the standard library of Python as the module ***re***. Note that there is also a the third-party ***regex*** module, with the API compatible with the ***re*** module, but offering additional functionality. You can find in on PyPI[3]'s page.

9.1.1 Metacharacters

Most characters used in a regex stand just for themselves. However, there are some characters that have special meaning; if you want your regex to contain such character literally, usually you have to escape it with backslash character '\'. The metacharacters are:

[1] https://docs.python.org/3/library/re.html#re-syntax

[2] https://docs.python.org/3/howto/regex.html

[3] https://pypi.org/project/regex

- . — dot;
- ^ — caret (“hat”);
- $ — dollar;
- * — asterisk;
- + — plus;
- ? — question mark;
- | — pipe;
- \ — backslash;
- (and) — parentheses;
- [and] — square brackets;
- { and } — braces.

Whether a character from the list above is special or not, sometimes depends on the context in which it appears. We will talk about this in what follows.

9.1.2 Classes

Classes define sets of characters — they are useful when we are looking not for a specific character, but rather for any character belonging to a set. They are specified in square brackets — a hyphen between characters denotes a range, a caret, or “hat”, ^, but only at the beginning, denotes negation. For example:

- `[abc]` — any of three letters: ‘a’, ‘b’ and ‘c’,
- `[a-d]` — any of four letters: ‘a’, ‘b’, ‘c’ and ‘d’,
- `[a-cu-z]` — any lowercase letter from ranges `[a-c]` or `[u-z]`,
- `[a-zA-Z]` — any lower- or uppercase Latin letter,
- `[a-zA-Z0-9_]` — any Latin letter, or a digit, or an underscore,
- `[^0-9]` — any character, but *not* a digit.

Regexes cannot be ANDed (as in Java), but they can be ORed with (single) symbol `|`: regex `cat|dog` will look for ‘cat’ OR ‘dog’.

Some metacharacters loose their special meaning inside sets, for example (,), +, *, . (dot). Inside a set they just stand for themselves. Also, classes such as `\w` or `\s` (see below) can be used inside classes; for example, as `\w` means *letter, digit, or underscore*, we can define a class `[\w.,]` which would match *any letter, digit, underscore, dot or comma*.

The character ^ denotes negation of the whole class, but only if it is used as the first character of a class; somewhere inside, it stands for itself. Therefore, `[^0-9]` means *anything, but* not *a digit*, while `[0-9^]` means *any digit or caret*. To match a literal] inside a set, just escape it with a backslash.

9.1.3 Predefined classes of characters

Some (unfortunately, not so many in Python) most useful classes are predefined and denoted by special symbols; usually it is just one letter after a backslash — in these cases uppercase letter means the same as the corresponding lowercase symbol, but negated. For example:

- `\d` — any digit, `\D` — any non-digit,
- `\s` — any white character (space, tab, new line), `\S` — anything except white character,

- `\w` — any letter, digit or underscore, `\W` — negation of `\w`: anything, but not letter, digit or underscore;
- `.` — any character except the new line (or including it, if DOTALL option has been selected, see sec. 9.4.2, p. 183),
- etc.,

Note: as Python works on UNICODE strings by default, *digit* means any digit, not necessarily Arabic; *letter* means any letter, in any language.

Unfortunately, *re* module in Python does not support some important and very useful predefined classes that we know from Java: `\p{L}` – any letter, in any language (but letters *only*, without underscores and digits), and its negation `\P{L}` (note capital 'P'), and also `\p{P}` – any punctuation mark, in any language, and its negation `\P{P}`.

We can manage without these symbols by defining our own versions of these classes:

```
import string
LETTER    = r'[^\W\d_]'       # any letter (no digits, no underscores)
NONLETTER = r'[\W\d_]'        # any non-letter
PUNCT     = '[' + re.escape(string.punctuation) + ']'   # any punct.
NONPUNCT  = '[^' + re.escape(string.punctuation) + ']' # any non-punct.
```

Here, we used string string.punctuation which contains punctuation marks

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Many of these punctuation marks are metacharacters, but we want to use them literally; the **re.escape** function returns the same string but with all metacharacters properly escaped, so inside a regex they will stand for themselves:

```
>>> import re
>>> import string
>>> re.escape(string.punctuation)
'!"\\#\\$%\\&\'\\(\\)\\*\\+,\\-\\./:;<=>\\?@\\[\\\\\\]\\^_`\\{\\|\\}\\~'
```

9.1.4 Special locations

There are symbols which denote not characters but rather special locations in the text being analyzed. For example:

- `\A` — beginning of the text;
- `^` — beginning of the text, but also of a line, if MULTILINE option is enabled – see sec. 9.4.1, p. 183;
- `\b` — word boundary — just before or just after a word (a word is a sequence of characters matching `\w`); `\B` — not at the beginning or end of a word;
- `$` — end of the text or of a line, if MULTILINE option is enabled;
- `\Z` — end of the text.

Important:
The slash character is extensively used in regular expressions. However, in string literals, slash is also special for Python interpreter, playing the rôle of the escape

character. That poses a problem. Suppose, we are looking for the sequence of three characters `'a\b'`. The regex engine needs to see *two* backslashes, as the signal, that this backslash is to be taken literally, and not as the escape character. So we have to include in our regex two literal backslashes. But to include a backslash into a literal string, we have to double it, because single backslash would be treated as the escape character by Python. So we end up with `'a\\\\b'`. Therefore, to avoid such monsters, when defining regexes, we almost always use raw strings, i.e., with letter `r` (or `R`) just before opening apostrophe or double quote (also in "tripled" form), as in these literal strings Python treats backslashes as "normal" characters (see sec. **??**, p. **??**). For example, `'\n'` is one-character string (new-line), while `r'\n'` contains two characters: backslash and the letter 'n'.

9.1.5 Quantifiers

You can put a so called *quantifier* just after an element of a regex. It then determines a possible number of repetitions of this element. For example:

- `+` — once or more;
- `*` — any number of occurrences (including zero);
- `?` — once or not at all;
- `{n,m}` — number of occurrences in the range $[n, m]$;
- `{n,}` — at least `n` occurrences;
- `{,m}` — at most `m` occurrences (perhaps zero).

All these quantifiers are by default **greedy**. It means that the regex engine will try to find the *longest* possible match. For example, if our regex is `a.*z` and the text is `"abzczdz"`, then the whole text will be found as the match, even though substrings `"abz"` and `"abzcz"` would be also possible (but are shorter).
If this is not what we want, we can make a quantifier **reluctant**, i.e., the engine will try to find the *shortest* match — in the above example `"abz"` would be found. To make a quantifier reluctant, we just add a question mark (?). Thus, continuing the above example, the regex `a.*?z` would find the shortest match (`"abz"`), while `a.*z` finds the longest (`"abzczdz"`).

Python does not support possessive quantifiers, which are supported in Java, but very rarely used.

9.1.6 Substitutes for methods using regexes of class *String* in Java

In Java, we have several methods of class **String** which use regexes and generally are extremely useful: **split**, **replaceAll** and **matches**. In Python, similar functionalities are provided by functions from the ***re*** module.

For example, suppose we want to split a text into words (sequences of letters, but only letters – no digits or underscores). Then we can assume that any non-empty sequence of non-letters may be treated as a separator between words. In Java, we would use the **split** method of the **String** class with separator `\P{L}+`; in Python we can use also the **split** function, but from the ***re*** module and get the same effect. There is no `\P{L}` class in Python, but, as we have already noted above, `r'[\W\d_]'` is the Python equivalent:

```
import re

```

```
3     NONLETTERs = r'[\W\d_]+' # any non-letter
4
5     s = ",,, __ 123 Łódź - 0.7; London - 8.8; Tokyo - 13.6"
6
7     ls = re.split(NONLETTERs, s)  # sequence of non-letters
8     print(ls)
9     ls = [i for i in re.split(NONLETTERs, s) if i]
10    print(ls)
```

prints

```
['', 'Łódź', 'London', 'Tokyo', '']
['Łódź', 'London', 'Tokyo']
```

In the sample string above, we have separators (sequences of non-letters) at the beginning, before the first "true" word, and also at the end, after the last one. As we can see, splitting produces in such situations empty elements as the first and the last element in the resulting list. The 9th line shows how we can get rid of them.[1]

As we know, regexes can also be ORed, as the following example illustrates: here, the separator is defined as non-empty sequence of white characters OR a punctuation mark surrounded, perhaps, but not necessarily, by sequences of white characters. The text itself is read from a file, line by line:

Listing 38 REA-Split/Splitting.py

```
#!/usr/bin/env python

import re
import string

pP = '[' + re.escape(string.punctuation) + ']'

sep = re.compile(r'\s*' + pP + '\s*|\s+')
with open('splitting1.txt', 'r') as f:
    for line in (line.rstrip() for line in f):
        words = sep.split(line)
        print('|' + "|".join(words) + '|')
```

If the file contains

```
John,Mary    Charles    ;    Zoe
carrot    parsley:potato
```

then the program prints

```
|John|Mary|Charles|Zoe|
|carrot|parsley|potato|
```

(the vertical bars were added to show the boundaries of words found).

[1] In Java, the first empty element would be included, but the trailing one would be discarded; unfortunately, conventions vary from language to language...

The second Java method, **String.replaceAll**, invoked on a string takes a regex and a replacement string and returns a string with all occurrences of substrings matching the regex replaced by the given replacement string. In Python, a similar function is called **sub** (as *sub*stitute); it takes a regex, replacement string and a text (string):

```
import re

string = "cat, caterpillar, tomcat, cat"
print( re.sub(r'\bcat\b', 'dog', string) )
```

prints `dog, caterpillar, tomcat, dog`, as we substitute `dog` for `cat`, but only if `cat` is surrounded by word boundaries, i.e., if `cat` is a separate word, not a substring of a word: note that `cat` in `tomcat` will not be replaced because there is no word boundary before the letter 'c', and similarly after the letter 't' in `caterpillar`.

The third method from Java that we want to replace with a corresponding Python construct is **String.matches**. It takes a regex and answers the question 'does the whole string match the regex'. In Python, we can use the **fullmatch** function from the ***re*** module: it returns the so called **match object**, which in Boolean context is "truthy" if the whole string matches the regex, and **None** (which is "falsy") otherwise:

```
import re

pLseq = r'[^\W\d_]+'        # sequence of letters

string = 'Madagascar!'
print('True' if re.fullmatch(pLseq, string) else 'False')
print('True' if re.fullmatch(pLseq+'.', string) else 'False')
```

prints

```
False
True
```

In the first case, entire string must be a sequence of only letters, which it isn't; in the second case, however, we allow for any character at the end, and this "any" matches the exclamation mark.

9.2 Pattern objects

Regexes, before being used, must be compiled. This is done by invoking the **compile** function, which returns an object of type **Pattern** which represents the compiled form of our regex:

```
>>> import re
>>> reg = r'[^\W\d_]+'      # sequence of letters
>>> pat = re.compile(reg)
>>> type(pat)
<class 're.Pattern'>
>>> str(pat)
"re.compile('[^\\\\W\\\\d_]+')"
```

As we can see, the __str__ method of Pattern gives us the original regex itself, although as a “normal”, not raw, string.

The compiled pattern is essentially a function (usually very complicated!) which, given a string, creates and returns an object holding information about the result of applying the regex to the string: this may be None, a list, a tuple, or a so called **match object** which can then be interrogated to get information we need.

In the examples presented so far, we didn’t compile regexes explicitly. Nevertheless, under the hood, they were compiled and the resulting pattern objects were used. This is acceptable, if a given regex is used once or twice. If you want to use it many times, however, you should always compile it once and then use the compiled pattern for different texts (strings); this is because compiling a regex is a complicated and expensive process.[1]

Many functions of the *re* module come in two flavors: as functions

```
re.fun(regex, string, ...
```

and as methods of class Pattern which are invoked on a compiled pattern (say, pat)

```
pat.fun(string, ...)
```

(we will see examples shortly).

9.3 Capturing groups

Capturing group allows us to remember parts of a substring matching a pattern, so that we can use it later. A group is created when a part of a regex is enclosed by round parentheses; some symbols immediately following the opening parenthesis modify the interpretation of groups, as we will discuss below.

9.3.1 Plain groups — (...)

Let us consider the following example:

Listing 39 REB-Groups/Groups.py

```
#!/usr/bin/env python

import re

LETTER    = r'[^\W\d_]' # any letter (no digits, no underscores)
NONLETTER = r'[\W\d_]'  # any non-letter

reg = f'({LETTER}+){NONLETTER}+({LETTER}+)'
#        <group 1>               <group 2>
print('reg: ' + str(reg))

string = 'John <_O_> Mary'
m = re.fullmatch(reg, string)

print('type of m: ' + str(type(m)))
```

[1] Python interpreter can remember some number of last regexes compiled and use them without recompilation.

```
16 print('m: ' + str(m))
17
18 print('m.group():        ' + m.group())
19 print('m.group(0):       ' + m.group(0))
20 print('len(m.groups()): ' + str(len(m.groups())))
21 print('m.lastindex:      ' + str(m.lastindex))
22 print('m.groups():       ' + str(m.groups()))
23 print('m.group(1):       ' + m.group(1))
24 print('m.group(2):       ' + m.group(2))
25 print('m.start(2):       ' + str(m.start(2)))
26 print('m.end(2):         ' + str(m.end(2)))
27 print('m.span(2):        ' + str(m.span(2)))
```

The regex defined in line 8 means *two words separated by a non-empty sequence of non-letters*. However, parts representing the words (but not the separating sequence of non-letters) are put into parentheses. There are two pairs of parentheses and whatever will match the regexes inside them will be remembered as group 1 and group 2.

On line 13, we call the **fullmatch** function, that we already know: it returns a *match object* (*matcher*) if the whole string matches the regex, and **None** otherwise. Our string (line 12) *does* match the regex, so m is such a match object.

The program prints

```
reg: ([^\W\d_]+)[\W\d_]+([^\W\d_]+)
type of m: <class 're.Match'>
m: <re.Match object; span=(0, 15), match='John <_0_> Mary'>
m.group():        John <_0_> Mary
m.group(0):       John <_0_> Mary
len(m.groups()): 2
m.lastindex:      2
m.groups():       ('John', 'Mary')
m.group(1):       John
m.group(2):       Mary
m.start(2):       11
m.end(2):         15
m.span(2):        (11, 15)
```

and illustrates several methods of match objects:

- **group()** and equivalent **group(0)** return the whole match;
- **groups()** returns a tuple of the contents of all groups; in our case two groups with matched sequences of letters;
- **group(n)** returns the content of the nth group (first group has index 1, index 0 means the whole match);
- **start(n)** returns position (index) of the first character of the nth group in the entire string;
- **end(n)** returns position (index) of the character following the last character of the nth group;
- **span(n)** returns both start and end as a tuple;
- lastindex is the index of the group which was closed as the last one — this is *not* necessarily the number of groups, because groups may be nested (see the next example); it is an attribute of the match object, not a method.

Groups may be nested and are numbered starting from 1, group number 0 being the whole substring matched. Numbers (indices) are assigned according to the order in which opening parentheses of the groups are encountered — each group extends to the corresponding closing parenthesis. For example, in the program below,

Listing 40 REC-Nesting/Nesting.py

```python
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

#            1                    2    3       opening
regex = r'\s*(' + LETTERs + r').*?(\d+-(\d+))'
#                             1            32  closing

print('regex: ' + str(regex))

string = "   Einstein Albert, 1879-1955"
m = re.fullmatch(regex, string)

print('m.group():       ' + m.group())
print('m.lastindex:     ' + str(m.lastindex))
print('len(m.groups()): ' + str(len(m.groups())))
print('m.groups():      ' + str(m.groups()))
print('m.group(1):      ' + m.group(1))
print('m.group(2):      ' + m.group(2))
print('m.group(3):      ' + m.group(3))
```

there are three groups:

- one containing the first word only, after, perhaps, some leading spaces, and followed by a sequence of any characters (which will not enter any group);
- then a group with two sequences of digits separated by a dash;
- then a group nested in the second and consisting of only the second of these two sequences of digits.

Note that the group *closed* as the last is group 2, not 3 — therefore, lastindex is 2 here and is not equal to the number of groups.

Note also that in this regex we used the reluctant regex `.*?` — without this question mark, the greedy `.*` would consume the first three digits of the first number, leaving only the fourth for the first `\d+` to swallow!

The program prints

```
regex: \s*([^\W\d_]+).*?(\d+-(\d+))
m.group():          Einstein Albert, 1879-1955
m.lastindex:     2
len(m.groups()): 3
m.groups():      ('Einstein', '1879-1955', '1955')
m.group(1):      Einstein
```

```
m.group(2):      1879-1955
m.group(3):      1955
```

9.3.2 Non-capturing groups — (?: ...)

Sometimes we use parentheses, e.g., to use ORing or apply a quantifier to a part of a regex, but it's not our intention to remember the matched substring as a group. Then, just after the opening parenthesis, we add a question mark and a colon, like this: `(?: ... )`. For example, in the following program

Listing 41 RED-NoGroup/NoGroup.py

```python
#!/usr/bin/env python

import re

LETTERs    = r'[^\W\d_]+' # sequence of letters

regex = r'(?:Mr\.|Mrs\.)\s+' + LETTERs + f'\\s+({LETTERs})'
#           not group                             group

print('regex: ' + str(regex))

string1 = 'Mr. Adam  Smith'
string2 = 'Mrs. Jane Gordon'

m1 = re.fullmatch(regex, string1)
print('m1.groups(): ', m1.groups())

m2 = re.fullmatch(regex, string2)
print('m2.groups(): ', m2.groups())
```

the first pair of parentheses doesn't create any group, it's used only to OR two options, `Mr.` or `Mrs.`. The first name must be there, but is not caught in any group either; we only catch the last name, so there is only one group here:

```
regex: (?:Mr\.|Mrs\.)\s+[^\W\d_]+\s+([^\W\d_]+)
m1.groups():  ('Smith',)
m2.groups():  ('Gordon',)
```

9.3.3 Named groups — (?P<name> ...)

Instead of numbering groups, we can assign names to them; the syntax is `(?P<name> ... )`, where `name` is any unique identifier (angle brackets are required). We then refer to such groups by invoking `matcher.group('name')`, instead of `matcher.group(index)` (although indices may still be used). Both ways of accessing groups, by index or by name, are illustrated in the program below

Listing 42 REE-NamedGroups/NamedGroups.py

```
 1 #!/usr/bin/env python
 2 
 3 import re
 4 
 5 regex1 = r'(\d{1,2})-(\d{1,2})-(\d{4}|\d{2})';
 6 regex2 = r'(?P<D>\d{1,2})-(?P<M>\d{1,2})-' \
 7                      r'(?P<Y>\d{4}|\d{2})'
 8 
 9 print('regex1: ' + str(regex1))
10 print('regex2: ' + str(regex2))
11 
12 string = '12-07-21 x_x 6-6-2010 123 y! 1-11-2011'
13 
14 matchiter = re.finditer(regex1, string)
15 print('type(matchiter): ' + str(type(matchiter)))
16 print('By indices:')
17 for matcher in matchiter:
18     print(f'{matcher.group(1)}/{matcher.group(2)}/'
19           f'{matcher.group(3)}', end='  ')
20 
21 matchiter = re.finditer(regex2, string)
22 print('\nBy names:')
23 for matcher in matchiter:
24     print(f'{matcher.group("D")}/{matcher.group("M")}/'
25           f'{matcher.group("Y")}', end='  ')
26 print()
```

which prints

```
regex1: (\d{1,2})-(\d{1,2})-(\d{4}|\d{2})
regex2: (?P<D>\d{1,2})-(?P<M>\d{1,2})-(?P<Y>\d{4}|\d{2})
type(matchiter): <class 'callable_iterator'>
By indices:
12/07/21  6/6/2010  1/11/2011
By names:
12/07/21  6/6/2010  1/11/2011
```

The two regexes look for a date, i.e., three numbers separated with a dash: one or two digits for day, one or two digits for month, and four or two digits for year. Note that in the last case we had to use `(\d{4}|\d{2})`, not the other way around (lines 5 and 7). Had we used `(\d{2}|\d{4})`, the first alternative, with two digits, would match the first two digits of a four-digit year, and `\d{4}` would not be even considered.

The first regex (line 5) doesn't use names, so we can refer to these groups later only by index (lines 18-19). In the second (lines 6-7), we give names 'D', 'M' and 'Y' to the corresponding groups, so we can refer to them by their names later in lines 24-25 (although indices would also work).

The **finditer** function, used in lines 14 and 21, returns an iterator which, when iterated over, yields a matcher object for each substring matching the regex (see sec. 9.5.7, p. 190).

9.3.4 Internal backreferences

Groups can also be referred to inside a regex itself. Using `\k` we can refer to the previously found group with index *k* (this is called **backreferencing**). We can also use `\g<k>` instead of `\k` — this is useful, if there is a digit right after a backreference. If the group was given a name, we can also refer to it by `(?P=name)`
In the following example we want to find fragments of a text enclosed by apostrophes or double quotes, but we have to ensure that the closing quote is of the same type as the opening one:

Listing 43 REF-BackRefs/BackRefs.py

```python
#!/usr/bin/env python

import re

regex1 = r'(["\'])([^"\']*)\1';
regex2 = r'(?P<quote>["\'])([^"\']*)(?P=quote)'

print('regex1: ' + str(regex1))
print('regex2: ' + str(regex2))

string = "'abc' xx \"def\" yy \"ghi' zz"

matchiter = re.finditer(regex1, string)
print('By indices:')
for matcher in matchiter:
    print(f'{matcher.group(1)}{matcher.group(2)}'
          f'{matcher.group(1)}', end='  ')

matchiter = re.finditer(regex2, string)
print('\nBy name or index:')
for matcher in matchiter:
    print(f'{matcher.group("quote")}{matcher.group(2)}'
          f'{matcher.group(1)}', end='  ')
print()
```

The regexes contain a group consisting of one character — either a single or a double quote — followed by a group of any number of any characters but excluding quotes of both types, followed by a quote matched in the first group: if it was a single quote, then the ending quote must also be single, if double, then this one has to be double as well. The program prints

```
regex1: (["\'])([^"\']*)\1
regex2: (?P<quote>["\'])([^"\']*)(?P=quote)
By indices:
'abc'  "def"
By name or index:
'abc'  "def"
```

Note that the sequence `"ghi'` was enclosed by non-matching quotes, so, correctly, it has not been found.

9.3.5 External backreferences

Backreferences to matched substrings can also be used in replacement texts when using the **sub** function, that we have already encountered (see p. 170). In a replacement text, we can refer to a matched group by its index using `\k` or `\g<k>`, or, if a group was given a name, with `\g<name>`.

In the following example, we use, just for illustration, not the **re.sub** function, as we did before, but the **sub** method of the **Pattern** class (see sec. 9.2, p. 170):

Listing 44 REG-Replace/Replace.py

```python
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

regex = f'(?P<first>{LETTERs})\\s+(?P<last>{LETTERs})'
print('regex: ' + str(regex))

patt = re.compile(regex)

strold = "John  Smith, Mary   Brown"

strnew = patt.sub(r'\2 \1', strold)
print(strold + '  ==>>  ' + strnew)
  # or
strnew = patt.sub(r'\g<2> \g<1>', strold)
print(strold + '  ==>>  ' + strnew)
  # or
strnew = patt.sub(r'\g<last> \g<first>', strold)
print(strold + '  ==>>  ' + strnew)
  # or mixed
strnew = patt.sub(r'\g<last> \1', strold)
print(strold + '  ==>>  ' + strnew)
```

The regex here means: two sequences of letters separated by a sequence of white characters. Both sequences of letters are taken into groups, named *first* and *last*, respectively. Then we replace all matches by the same sequences, but in reverse order and separated by exactly one space. We do it in four equivalent ways, using indices and names; the result is the same:

```
regex: (?P<first>[^\W\d_]+)\s+(?P<last>[^\W\d_]+)
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
```

9.3.6 Lookahead and lookbehind assertions — (?= ...), (?! ...), (?<= ...), (?<! ...)

Sometimes we want to find a substring matching a regex, but only in some context: for example only, if the match is followed or preceded by a match for something else, or just opposite, when it's *not* followed or preceded by a match for something else. We call it positive or negative **lookahead assertions** or positive or negative **lookbehind assertions**;

- (?= ...) — positive lookahead assertion,
- (?<= ...) — positive lookbehind assertion,
- (?! ...) — negative lookahead assertion,
- (?<! ...) — negative lookbehind assertion.

Lookaheads and lookbehinds are zero length assertions, that means they are not included in the match. They only assert whether immediate portion after or before a given input string's current portion is suitable for a match or not.

In the example below, we use the **finditer** function, returning an iterator which, when iterated over, yields a matcher object for each substring matching the regex (see sec. 9.5.7, p. 190).

Listing 45 REH-Look/Look.py

```python
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

string = 'carrot 5.7, butter 6.5, carrot 5 milk ' \
         'carrot 7.3  parsley 3.5 carrot 2.00  ' \
         'butter 2 carrot apples'
# -- 1 --
# sum of prices only of carrot with a price
regex = r'(?<=carrot) (\d+(?:\.\d+)?)'
matchiter = re.finditer(regex, string)
summ = 0
for matcher in matchiter: summ += float(matcher.group(1))
print('carrot:              ' + str(summ))

# -- 2 --
# sum of prices of all items with a price, except carrot
regex = r'(?<!carrot) (\d+(?:\.\d+)?)'
matchiter = re.finditer(regex, string)
summ = 0
for matcher in matchiter: summ += float(matcher.group(1))
print('all but carrot:      ' + str(summ))

# -- 3 --
# set of items with price indicated
regex = f'({LETTERs})\\b(?=(?: \\d+))'
```

```
29 matchiter = re.finditer(regex, string)
30 sett = set()
31 for matcher in matchiter: sett.add(matcher.group(1))
32 print('items with price:    ' + str(sett))
33
34 # -- 4 --
35 # set of items with price not indicated
36 regex = f'({LETTERs})\\b(?!(?: \\d+))'
37 matchiter = re.finditer(regex, string)
38 sett = set()
39 for matcher in matchiter: sett.add(matcher.group(1))
40 print('items with no price: ' + str(sett))
```

Let us look at the first example here. In the regex on line 12, there is only one group: `(\d+(?:\.\d+)?)`. It means *a sequence of digits, optionally followed by a dot and another sequence of digits*. This optional expression is not a group, because of `?:` at the beginning — it only indicates to what part of the regex the '?' is to be applied. Before the group we have a space, and before this space there must be the word *carrot*, which, however, is *not* a part of the match — this is indicated by the preceding positive lookbehinds. In other words, we are looking for numbers, but only following `carrot`.

The second example is similar, but now we have a negative lookbehind — before the number, there cannot be `carrot`.

The third and fourth examples illustrate lookaheads: we construct sets of names of products that are or just opposite, are not, followed by a price.

The program prints

```
carrot:             20.0
all but carrot:     12.0
items with price:   {'butter', 'parsley', 'carrot'}
items with no price: {'milk', 'apples', 'carrot'}
```

And the final example: the program below uses lookaheads to find out whether a given string can be a valid password. Here, we assume that a password should be at least 8 characters long and contain at least one capital letter, one lower-case letter, one digit and one special character (in the program it is one of the character of the PC string):

Listing 46 RER-PassCheck/PassCheck.py

```
#!/usr/bin/env python

import re

def checkPass(password):
    PC =r'!@#$%^&*><?.,;:' # special characters
    p1 = (fr'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)'
          fr'(?=.*[{PC}])[A-Za-z\d{PC}]{{8,}}$')
      # or, equivalently with verbose mode on
    p2 = fr'''(?x)^             # verbose; start of string
              (?=.*[A-Z])       # capital letter somewhere
              (?=.*[a-z])       # lower-case letter somewhere
              (?=.*\d)          # digit somewhere
              (?=.*[{PC}])      # special character somewhere
              [A-Za-z\d{PC}]{{8,}}$  # at least 8 chars, EOS
           '''
    return re.match(p2, password)

for pas in [r'Ab?De93',  r'A1b:A1b>',  r'Ab/Acb<1',
                 r'abc123><', r'Zorro@123', r'ab<AB!1z']:
    print(f'{pas:9} is {"" if checkPass(pas) else "not "}valid')
```

Regexes p1 and p2 are equivalent; in the latter, we used verbose mode (see sec. 9.4.5, p. 185) to make the meaning easier to grasp. The program prints

```
Ab?De93   is not valid      # too short
A1b:A1b>  is valid
Ab/Acb<1  is not valid      # / is not among special chars
abc123><  is not valid      # no capital letter
Zorro@123 is valid
ab<AB!1z  is valid
```

(note double braces in {{8,}}; in f-strings, braces have special meaning and to get one literal brace, we have to double it).

9.3.7 Yes-no patterns

The syntax of this rarely used construct is

```
(?(id/name)pattern_if_yes|pattern_if_no)
```

This will try to match either `pattern_if_yes` or `pattern_if_no`, depending on whether group referred to by `id` (1, 2, . . .) or by `name` (for named groups) was matched or not. For example,

Listing 47 REI-YesNo/YesNo.py

```
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

lines = ['#bananas',
         '255',
         '#tomatos',
         '4',
         '#apples',
         '30']

regex = f'(#)?((?(1){LETTERs}|\\d+))'
patt = re.compile(regex)
for line in lines:
    print( str( patt.fullmatch(line).group(2) ) )
```

will try to match a word or a number, depending on whether at the beginning there was character '#' or not

```
bananas
255
tomatos
4
apples
30
```

9.3.8 Comments — (?# ...)

Finally there are groups which are in fact not only not groups (as `(?: ...)`, but are ignored completely. They can serve just as comments and are not processed by the regex engine at all. For example:

```
import re
LETTERs = r'[^\W\d_]+' # sequence of letters

regex = f'{LETTERs}(?# letters and then)\\.(?# a dot)'
string = 'Mississippi.'
m = re.fullmatch(regex, string)
if m: print(m.group())
```

prints `Mississippi.`.

9.4 Option flags

There are several options which influence the process of compilation of a pattern. They can be specified in two ways:

- as an additional argument to the **re.compile** function, or to other functions from the ***re*** module which take a non-compiled regex (just string) as the first argument — as we know, under the hood, the compilation will take place anyway; or,
- they can be set directly inside the string representing a regex (the so called *inline flag*).

Options are defined in the ***re*** module as constants of the **RegexFlag** enumeration (which is a subclass of **enum.IntFlag**) and may be ORed with a vertical bar |, if we want to set several of them.

If we choose to specify them inlined in our regex, we do it by including expression `(?letters)` where `letters` can be one or more letters denoting different options.

For example, suppose we want to turn on the compilation options DOTALL and MULTILINE; we can do it like this

```
>>> import re
>>> regex = "..."
>>> regObj = re.compile(regex, re.MULTILINE | re.DOTALL)
```

or like this

```
>>> import re
>>> regex = "..."
>>> regObj = re.compile(regex, re.M | re.S)
```

because re.MULTILINE is equivalent to re.M and re.DOTALL is equivalent to re.S.
But we can also specify the required options by embedding `(?letters)` directly into the regex itself

```
>>> import re
>>> regex = "(?sm)..."
>>> regObj = re.compile(regex)
```

as the letter 's' denotes DOTALL ('s' because it's called 'single-line' in Perl) and 'm' stands for MULTILINE.
Note that enabling options usually induces some performance penalty, so don't do it if it's not necessary.

As options flags are constants of an enumeration, they correspond to integers (and can be ORed). The program below shows all flags, their full and abbreviated names, the corresponding letters that can be used in the inline form, and integers that they correspond to:

Listing 48 REK-RegFlags/RegFlags.py

```
#!/usr/bin/env python

import re

regex  = r'\d+'
patObj = re.compile(regex, flags=0)
print('Default flags: ' + str(patObj.flags) + '\n')
print('Type of flags: ' + str(type(re.ASCII)) + '\n')

print(f'IGNORECASE I (?i) -> {int(re.I):3d}')
print(f'LOCALE     L (?L) -> {int(re.L):3d}')
print(f'MULTILINE  M (?m) -> {int(re.M):3d}')
print(f'DOTALL     S (?s) -> {int(re.S):3d}')
print(f'UNICODE    U (?u) -> {int(re.U):3d}')
print(f'VERBOSE    X (?x) -> {int(re.X):3d}')
print(f'DEBUG             -> {int(re.DEBUG):3d}')
print(f'ASCII      A (?a) -> {int(re.A):3d}')
```

The program prints

```
Default flags: 32

Type of flags: <enum 'RegexFlag'>

IGNORECASE I (?i) ->   2
LOCALE     L (?L) ->   4
MULTILINE  M (?m) ->   8
DOTALL     S (?s) ->  16
UNICODE    U (?u) ->  32
VERBOSE    X (?x) ->  64
DEBUG             -> 128
ASCII      A (?a) -> 256
```

The UNICODE flag is not used, as Unicode is now the default — it's still present in the language for compatibility with previous versions of Python. The DEBUG flag, when set, is used for debugging — the program will print information about the process of compiling a regex; the output is understandable for experts only, so we will not discuss it. Also LOCALE is rarely used, as it can only be applied to byte patterns, not "normal" strings.

The remaining flags are described below.

9.4.1 *MULTILINE*

Abbreviated form of re.MULTILINE is re.M. In embedded (inline) form it corresponds to the letter 'm'.

This option enables the so called 'multi-line mode', which means that metacharacters `^` and `$` match at the beginning of each a line and at the and of each line, respectively. Normally, when this option is *not* enabled, these symbols match only at the beginning and at the end of the entire input text (but in multi-line mode, those are still available as `\A` and `\Z`).
For example, the following program

Listing 49 REL-FlagM/FlagM.py

```python
#!/usr/bin/env python

import re

string = "A 123\nD 456";

matchiter1 = re.finditer(r'^\w',     string)
matchiter2 = re.finditer(r'(?m)^\w', string)

for matcher in matchiter1:
    print(f'{matcher.group()}', end=' ')
print()

for matcher in matchiter2:
    print(f'{matcher.group()}', end=' ')
print()
```

will print

```
A
A D
```

because in the first case we only looked for a letter at the beginning of the entire input, while in the second case — at the beginning of each line separately.

9.4.2 *DOTALL*

Abbreviated form of re.DOTALL is re.S. In embedded (inline) form it corresponds to the letter 's'.

This option changes the interpretation of the dot (.). By default, it denotes ‘any character except new line’, while with this option enabled a dot matches any character, *including* the new line.

For example, the following program

```
import re

string = "A 123\n456 B";
print('Matches' if (re.fullmatch(r'\w.*\w', string))
                else "Doesn't match")
print('Matches' if (re.fullmatch(r'\w.*\w', string, re.S))
                else "Doesn't match")
```

will print

```
Doesn't match
Matches
```

because in the first case `.*` consumes everything up to the new line (but not any further) and there is no match to the whole text, while in the second case everything, including the new line character, will be consumed by `.*` reaching the final letter ’B’.

9.4.3 *IGNORECASE*

Abbreviated form of re.IGNORECASE is re.I. In embedded (inline) form it corresponds to the letter ‘i’.

Enabling makes the lower- and uppercase letters indistinguishable.

For example, the following program

```
import re

string = "PaRiS";
print('Matches' if (re.fullmatch('Paris', string))
                else "Doesn't match")
print('Matches' if (re.fullmatch('Paris', string, re.I))
                else "Doesn't match")
```

will print

```
Doesn't match
Matches
```

as in the second case we ignore the case of letters, so `PaRiS` and `Paris` are considered equivalent.

9.4.4 *ASCII*

Abbreviated form of re.ASCII is re.A. In embedded (inline) form it corresponds to the letter ’a’.

Enabling this option makes `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching, instead of full Unicode matching. For example, `\w` matches the underscore, Arabic digits and ASCII (Latin) letters only, but *not* non-ASCII letters.

For example, the following program

```
import re

string = "Żółć Bałtyk Paris";
regex  = r'[^\W\d_]+' # sequence of letters

matcher1 = re.finditer(regex, string)
matcher2 = re.finditer(regex, string, re.ASCII)

print([m.group() for m in matcher1])
print([m.group() for m in matcher2])
```

will print

```
['Żółć', 'Bałtyk', 'Paris']
['Ba', 'tyk', 'Paris']
```

as in the second case `[^\W\d_]+` matches only sequences of Latin letters, and non-ASCII letters, like Polish 'Ż', 'ó', 'ł', 'ć', are *not* considered to be letters.

9.4.5 *VERBOSE*

Abbreviated form of re.VERBOSE is re.X. In embedded (inline) form it corresponds to the letter 'x'.

This flag is sometimes used just for convenience. When it is enabled

- white spaces (also new-line characters) in regex are ignored, except when they appear in a character class, or are preceded by a backslash;
- if there is a `#` character (not in a class and not preceded by a backslash), all characters from `#` through the end of the line are ignored.

This allows us to write regular expressions in a more readable form and add comments, as illustrated in the program

Listing 50 REM-Verbose/Verbose.py

```
#!/usr/bin/env python

import re

string = '2/11/2021 xxx 13-01/1999 yyy 13-10 zzz 21.2.2022'

regex1 = r'\d{1,2}([-/.])\d{1,2}\1\d{4}'
ls1 = re.finditer(regex1, string)
print([m.group() for m in ls1])

# now the same thing with VERBOSE

regex2 = r'''(?x)\d{1,2} # verbose; day: one or two digits
             ([-/.])     # day-month separator: - or / or .
             \d{1,2}     # month: one or two digits
             \1          # the same separator as day-month
             \d{4}       # year: four digits
```

```
            '''
ls2 = re.finditer(regex2, string)
print([m.group() for m in ls2])
```

which prints

```
['2/11/2021', '21.2.2022']
['2/11/2021', '21.2.2022']
```

For another example — see Listing 46.

9.5 Functions operating on regular expressions

Many function operating on regular expressions come in two forms:

- as functions from the *re* module, taking a regex as a string;
- as methods of the re.Pattern class, invoked on an already compiled regex.

In the first case, behavior of compilation can be modified by specifying flags (see sec. 9.4, p. 181) passed as arguments to these functions; of course, it's not possible in the second case, as here we are working on an already compiled regex.

.

Below, such pairs of functions will be described together: **re.function(...)** is understood to be a function from the *re* module, while **regObj.function(...)** is a corresponding method invoked on regObj, where object regObj of type Pattern is the compiled version of a regex string. Note that in the documentation, regex strings are called *regex patterns*, while their compiled versions – *regex objects*.

Functions **search**, **match** and **fullmatch** (and also **finditer**) return the so called *match objects* (or just *matchers*) of class re.Match. Several useful methods of this object can then be used to extract information about the result of applying a regex to a string (text) — they are also shortly described below, as are a few fields of the Pattern class.

9.5.1 re.compile

`re.compile(regex, flags=0)` — compiles a regex string into a regex object.

```
>>> regex = '(\d+) ([^\W\d_]+)'
>>> regObj = re.compile(regex, re.I)
>>> type(regObj)
<class 're.Pattern'>
```

Examples: ?? (p. ??), Listing 44 (p. 177), Listing 47 (p. 180) and Listing 48 (p. 182).

9.5.2 re.search

`re.search(regex, string, flags=0)` — searches for the first (and only the first!) substring of string matching the regex. Returns a corresponding match object, or None if there was no such substring.

The second version, as a method `regObj.search(string[,pos[,endpos]])`, has two additional, optional arguments:

- pos is an index in the string where the search is to start (default is 0);

- endpos limits how far the string will be searched (default is the length of the string); only the characters from pos to endpos-1 (inclusive) will be searched for a match.

For example,

```
import re
regex = r'\d+'
string = 'abc 123456789'
regObj = re.compile(regex)
m = regObj.search(string, 0, 7)
print(type(m))
print(m.group())
print(m.span())
```

prints

```
<class 're.Match'>
123
(4, 7)
```

because the first matching substring is ’123’, as we confined the search to the part of the string composed of characters on positions from 0 to 6 (inclusive). Here, m is the matcher object. We used its method **group()** which returns, as a string, the substring matching the regex; **span()** returns, as a tuple, the range of indices where this substring is located in the whole input string (the start index is *inclusive*, the end index is *exclusive!*).

9.5.3 re.match

re.match(regex, string, flags=0) — like **search**, but looks for a match *only* at the very beginning of the string.
The second version, as a method regObj.match(string[,pos[,endpos]]), looks for a match *only* at the very beginning of the substring of string starting at position pos (and ignoring the part starting at endpos). For example,

```
import re

regex = r'\d+'
string = 'abc 123456789'

regObj = re.compile(regex)
m = regObj.match(string)
print(m)

m = regObj.match(string, 6)
print(m.group())
print(m.span())
```

prints

```
None
3456789
(6, 13)
```

In the first case no match has been found, because a sequence of digits, although present, does not start at position 0. In the second case, we start from position 6 (character '3'), and now there *is* a sequence of digits starting from this position.

9.5.4 re.fullmatch

`re.fullmatch(regex, string, flags=0)` — like **match**, but this time the *whole* string must match the regex, a substring is not enough. Returns a matcher object if successful, and None otherwise.

There is also the corresponding method `regObj.fullmatch(string[,pos[,endpos]])` which works only on a substring defined by indices pos and endpos (if specified).

Examples can be found in Listing 39 (p. 171), Listing 40 (p. 173), Listing 41 (p. 174) and Listing 47 (p. 180).

9.5.5 re.split

`re.split(regex, string, maxsplit=0, flags=0)` — splits the string treating parts matching the regex as separators between "words"; returns the list of these "words". An example was already given in Listing 38 (p. 169).

Note: Using groups in a regex determining separators is not recommended, as it leads to non-intuitive results: the text of all captured groups will be added to the list. For example,

```
import re

regex = r'(:|\||;)'   #  ':' '|' and ';' are separators
string = ':ab|c;|3:'

lst = re.split(regex, string)
print(lst)
```

prints

```
['', ':', 'ab', '|', 'c', ';', '', '|', '3', ':', '']
```

Without any groups

```
import re

regex = r':|\||;'   #  ':' '|' and ';' are separators
string = ':ab|c;|3:'

lst = re.split(regex, string)
print(lst)
```

the program prints

```
['', 'ab', 'c', '', '3', '']
```

As we can see, if there is a separator at the beginning, we get an empty string as the first element of the resulting list; similarly, a separator at the end of the string gives us an empty string as the last element of the list. Also, there are two separators between 'c' and '3'; we also get an empty element in the list.

Additional argument maxsplit determines maximum number of splits: the resulting list will contain at most maxsplit+1 elements and after the first maxsplit "words", all the remaining text will go into the last, maxslit+1th element:

```
import re

regex = r'[\W\d_]+' # sequence of non-letters
string = 'John, Mary Adam,  Kitty Sue   Bill'

lst = re.split(regex, string, maxsplit=3)
print(lst)
```

prints

```
['John', 'Mary', 'Adam', 'Kitty Sue   Bill']
```

There is also the corresponding method of class **Pattern**: `regObj.split(string, maxsplit=0)`.

9.5.6 re.findall

`re.findall(regex, string, flags=0)` — returns the list of all substrings of `string` matching the regex. However, the result depends on the number of capturing groups in the regex: the function returns:

- if there are no groups, a list of strings matching the whole regex;
- if there is one group, a list of substrings matching that group;
- if there are two or more groups, a list of tuples of substrings matching these groups.

This is illustrated by the following example:

```
import re

string = 'John weight =  75, height= 182  Kitty 23'

# 1
regex = r'[^\W\d_]+\s*=\s*\d+'
print( re.findall(regex, string) )

# 2
regex = r'[^\W\d_]+\s*=\s*(\d+)'
print( re.findall(regex, string) )

# 3
regex = r'([^\W\d_]+)\s*=\s*(\d+)'
print( re.findall(regex, string) )
```

which prints

```
['weight =  75', 'height= 182']
['75', '182']
[('weight', '75'), ('height', '182')]
```

The regexes mean *a sequence of letters followed by an equal sign possibly surrounded by sequences of white spaces, followed by a sequence of digits*.

In the first case, there are no groups, so just substrings matching the regex are returned. In the second case, there is one group surrounding the sequence of digits, so only content of this group for each match is returned.

In the third case, both the sequence of letters and the sequence of digits are taken into groups. Then the resulting list contains *tuples* of substrings matching the groups.

The second version, as a method `regObj.findall(string[,pos[,endpos]])`, looks for a match *only* in the fragment of the text string starting at position pos and ignoring the part starting at endpos.

9.5.7 re.finditer

`re.finditer(regex, string, flags=0)` — returns an iterator yielding match objects for all non-overlapping matches for the regex. For examples, see Listing 42 (p. 175), Listing 43 (p. 176) and Listing 45 (p. 178).
The second version, as a method `regObj.finditer(string[,pos[,endpos]])`, looks for a match *only* in the fragment of the text string starting at position pos and ignoring the part starting at endpos.

9.5.8 re.sub

`re.sub(regex, repl, string, count=0, flags=0)` — returns the string obtained by replacing occurrences of substrings of string which match the regex by the replacement text repl. If no such substring was found, string is returned unchanged. The optional argument count is the maximum number of substrings to be replaced; if omitted or zero, all occurrences will be replaced.

In repl, we can use backreferences to groups, both by index (like \2 or \g<2>) and also, for named groups, by name (like \g<name>) — see sec. 9.3.5, p. 177.

The argument repl is usually just a string — possibly containing backreferences — but can also be a function. If so, this function will be called for every matching substring. The function should take one argument, a match object corresponding to the matched fragment of the text, and it should return a string which will then be used as the replacement text.

For an example illustrating the **sub** function, see Listing 44, p. 177. Another example, below, shows how to use a function instead of a replacement text:

Listing 51 REJ-SubFun/SubFun.py

```python
#!/usr/bin/env python

import re

def classify(matcher):
    g = matcher.group()
    num = int(g)
    if   num > 10: g += ' (too high)\n'
    elif num <  0: g += ' (too low)\n'
    else:          g += ' (OK)\n'
    return g

regex = r'-?\d+'
strold = '-7 11 5 2 12 9 -1'

strnew = re.sub(regex, classify, strold)
```

```
print(strnew)
```

The regex is very simple: *sequence of digits optionally preceded by a minus sign*. The **classify** function will be called for every match found; the corresponding match object will be passed to it and the matching string, but with some information added to it, will be returned — this string will be then used to perform actual replacement:

```
-7 (too low)
11 (too high)
5 (OK)
2 (OK)
12 (too high)
9 (OK)
-1 (too low)
```

The second version, as a method of class **Pattern** is, of course `regObj.sub(repl, string, count=0)`.

9.5.9 re.subn

`re.sub(regex, repl, string, count=0, flags=0)` — as **sub**, but returns a two-element tuple consisting of the string that would be returned by **sub** and, as the second element, number of replacements made. Alternatively, the method of class **Pattern** `patObj.sunb(repl, string, count=0)` may be used.

For example,

```
import re

string = 'cat, caterpilar,bulldog - dog, cat'
regex = r'\b(cat|dog)\b'

res = re.subn(regex, 'pet', string)
print(type(res))
print(res)
```

prints

```
<class 'tuple'>
('pet, caterpilar,bulldog - pet, pet', 3)
```

9.5.10 re.escape

`re.escape(regex)` — returns a string with all metacharacters in regex properly escaped, so it may be used as a regular expression string. Suppose, for example, that we have a TeX or LaTeX file, and we want to search for the literal string `\index{Sherlock}`. Instead of thinking hard how to escape all metacharacters in this string (i.e., `\`, `{` and `}`), we can just call **escape** which will do it for us:

```
>>> import re
>>> regex = r'\index{Sherlock}'
>>> re.escape(regex)
'\\\\index\\{Sherlock\\}'
```

or

```
>>> import re
>>> operators = r'Operators: *, /, +, -'
>>> re.escape(operators)
'Operators:\\ \\*,\\ /,\\ \\+,\\ \\-'
```

For another example, see p. 166.

9.5.11 re.purge

`re.purge()` — clears the regular expression cache. Python manages a special cache that is used internally to store compiled regular expressions; when the interpreter detects that the same regex is used again, it can use an already compiled object from the cache instead of compiling it anew. The caching mechanism is very effective and doesn't consume significant amount of memory, so normally clearing the cache doesn't serve any useful purpose.

9.6 Matcher methods

> *Below, we describe methods of the matcher object (of type* **re.Matcher***); they are invoked on a matcher returned by such functions as* **search**, **match**, **fullmatch** *and* **finditer**. *The methods will be illustrated by the following example:*

Listing 52 REN-Matcher/Matcher.py

```python
#!/usr/bin/env python

import re

string = 's23453 (Jane Brown)'

regex = r'''(?x)            # verbose
            (?P<snum>\d+)   # group 'snum': student number
            \s+\(           # spaces and (
            (?P<full>       # group 'full': full name
             [^\W\d_]+\s+
             (?P<last>      # group 'last': last name
              [^\W\d_]+
             )              # end of group 'last'
            )               # end of group 'full'
            \)
         '''

matchiter = re.finditer(regex, string)
for matcher in matchiter:
    print(' 0. ' + str(matcher.groups()))  # tuple of groups
    print(' 1. ' + matcher.group())
    print(' 2. ' + matcher.group(0))       # group(0) = group()
    print(' 3. ' + matcher.group(1))
    print(' 4. ' + matcher.group(2))
    print(' 5. ' + matcher.group(3))
    print(' 6. ' + matcher[3])             # group(i) = matcher[i]
    print(' 7. ' + matcher.group('snum'))  # group by name
    print(' 8. ' + matcher.group('last'))
    print(' 9. ' + str(matcher.group(1,3)))  # yields a tuple
    print('10. ' + str(matcher.groupdict())) # dict of named groups
    print('11. ' + str(matcher.start(1)))    # where group 1 starts
    print('12. ' + str(matcher.end(1)))      # and where it ends
    print('13. ' + str(matcher.start('last'))) # now by name
    print('14. ' + str(matcher.end('last')))
    print('15. ' + str(matcher.span(2)))       # (start, end) tuple
    print('16. ' + str(matcher.span('full')))
    print('17. ' + str(matcher.lastindex))     # last group closed
    print('18. ' + matcher.lastgroup)          # name of it
    print('19. ' + matcher.string)             # original text
    print('20. ' + str(type(matcher.re)))      # pattern object
```

There are three (named) groups here: the first catches the student number, the second his/her full name, and the third, embedded into the second, only the last name. The program prints

```
 0. ('23453', 'Jane Brown', 'Brown')
 1. 23453 (Jane Brown)
 2. 23453 (Jane Brown)
 3. 23453
 4. Jane Brown
 5. Brown
 6. Brown
 7. 23453
 8. Brown
 9. ('23453', 'Brown')
10. {'snum': '23453', 'full': 'Jane Brown', 'last': 'Brown'}
11. 1
12. 6
13. 13
14. 18
15. (8, 18)
16. (8, 18)
17. 2
18. full
19. s23453 (Jane Brown)
20. <class 're.Pattern'>
```

and these line numbers from the program and from the printout will be referred to in the explanations below.

9.6.1 matcher.groups, matcher.group

- `matcher.groups()` — returns a tuple of strings representing parts of the input string matching the groups defined in the regex; see line no 0. Number of groups is therefore `len(matcher.groups())`.

- `matcher.group()` — returns the part of the input string matching the regex (even if there were not groups defined in it); see line no 1.
- `matcher.group(index)` — returns, as a string, the group number index from the matched part of the input string. Of course, such a group must have been defined in the regex. With index equal to 0, the call is equivalent to calling just **group()** without arguments. See lines no 2-5.
- `matcher.__getitem__(index)` — is equivalent to `matcher.group(index)`. Note, that __getitem__ is called when an index is used (in square brackets), so normally you can write just `matcher[index]`, as illustrated in line no 6.
- `matcher.group('name')` — returns, as a string, the group named name from the matched part of the input string. Of course, such a group must have been defined in the regex and given a name. See lines no 7-8.
- `matcher.group(index1, index2, ...)` — returns, as a tuple of strings, matching substrings from groups with given indices — see line no 9.

9.6.2 matcher.groupdict

`matcher.groupdict()` — returns a dictionary with names of groups as keys and the corresponding matching substrings as values (only named groups are considered) — see line no 10.

9.6.3 matcher.start, matcher.end, matcher.span

- `matcher.start(index)` — returns the index in the entire input string, where the match caught in group no index starts; see line no 11.
- `matcher.end(index)` — returns the index in the entire input string of the first character *after* the last character of the match caught in group no index; see line no 12.
- `matcher.start('name')` — returns the index in the entire input string, where the match caught in group named name starts; see line no 13.
- `matcher.end('name')` — returns the index in the entire input string of the first character *after* the last character of the match caught in group named name; see line no 14.
- `matcher.span(index)` — returns a tuple containing indices `matcher.start(index)` and `matcher.end(index)`; see line no 15.
- `matcher.span('name')` — the same, but using the name of a group; see line no 16.

9.6.4 matcher.lastindex, matcher.lastgroup, matcher.re, matcher.string

These are not functions/methods, but rather fields (attributes) of the matcher object:

- matcher.lastindex — index of the group which was closed as the last. In our example this is 2, although there are three groups, because the second group was closed *after* the third; see line no 17.
- matcher.lastgroup — the name of the group which was closed as the last (it must have been given a name, of course). In our example this is full, although there are three groups, because the second group was closed *after* the third (named last); see line no 18.
- matcher.string — input string passed to **search**, **match**, **fullmatch** or **finditer** which this matcher comes from; see line no 19.

- matcher.re — the pattern object (i.e., compiled version of the regex) used to create this matcher — see line no 20.

9.7 Pattern object attributes

> *We have already described methods of pattern objects (see sec. 9.5, p. 186) but these objects have also some useful attributes. They will be illustrated by the following example, where the regex is the same as in the previous example (Listing 52).*

Listing 53 REO-PattAttr/PattAttr.py

```
 1  #!/usr/bin/env python
 2
 3  import re
 4
 5  regex = r'''(?P<snum>\d+)   # group 'snum': student number
 6              \s+\(           # spaces and (
 7              (?P<full>       # group 'full': full name
 8               [^\W\d_]+\s+
 9               (?P<last>      # group 'last': last name
10                [^\W\d_]+
11               )              # end of group 'last'
12              )               # end of group 'full'
13              \)
14           '''
15  patObj = re.compile(regex, re.VERBOSE)
16
17  print(' 1 -> ' + str(patObj.flags))       # flags (as an int)
18  print(' 2 -> ' + str(patObj.groups))      # number of groups
19  print(' 3 -> ' + str(patObj.groupindex))  # dict of (name, index)
20  print(' 4 -> ' + patObj.pattern)          # regex (as string)
```

The program prints

```
 1 -> 96
 2 -> 3
 3 -> {'snum': 1, 'full': 2, 'last': 3}
 4 -> (?P<snum>\d+)   # group 'snum': student number
            \s+\(           # spaces and (
            (?P<full>       # group 'full': full name
             [^\W\d_]+\s+
             (?P<last>      # group 'last': last name
              [^\W\d_]+
             )              # end of group 'last'
            )               # end of group 'full'
            \)
```

and line numbers from the program and from the printout will be referred to in the explanations below.

- patObj.flags — options flags used when compiling the regex; this is an integer which is equal to the ORed flags which were set. Here, we got 96, which is 32 (corresponding to re.UNICODE, set by default) plus 64 (corresponding to re.VERBOSE) — this can be seen from the output of the Listing 48. See line 1.
- patObj.groups — number of groups defined in the regex; see line no 2.
- patObj.groupindex — a dictionary with names of all named groups as keys and the corresponding indices as values; see line no 3.
- patObj.pattern — the regex used to create this pattern object; see line no 4.

Chapter 10

Exceptions

10.1 Raising and handling exceptions

Exceptions and exception handling provide means to change program flow when an unexpected or abnormal event has been encountered which prevents the program from normal continuation. This may be, for example, an attempt to divide by zero or to read from a file which doesn't exist. Exceptions are ubiquitous in Python, and it is therefore crucial to understand how to handle them.[1]

In situations in which the interpreter cannot continue normally, it "raises" an exception, which, as everything in Python, will be represented by an object (we know it also from C++/Java). If we don't tell the interpreter what to do when this happens, the program will crash, but at least it will display some information about the reason and location of the failure. Let us try:

```python
def g(num, d):
    return num // d

def f(num, a, b):
    for d in range(a, b+1):
        x = g(num, d)
        print(f'{num}//{d}={x}')

def main():
    f(5, -3, 2)

main()
```

Running this little program prints

```
5//-3=-2
5//-2=-3
5//-1=-5
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py", line 578,
    in runcode exec(code, self.locals)
File "/home/werner/p.py", line 12, in <module>
    main()
File "/home/werner/p.py", line 10, in main
    f(5, -3, 2)
File "/home/werner/p.py", line 6, in f
    x = g(num, d)
File "/home/werner/p.py", line 2, in g
    return num // d
ZeroDivisionError: integer division or modulo by zero
```

We read it from the bottom up

[1] Luckily, it is quite similar to what we already know from Java and C++. However, there is no notion of checked and unchecked exceptions, as in Java — in this sense, all exceptions are unchecked: we can, but we don't have to handle them.

- The exception that happened is represented by an object of type **ZeroDivisionError** with descriptive message *integer division or modulo by zero.*
- It was raised when executing line 2, in the body of the function **g**: the offending line was `return num // d`
- As the exception has not been handled, we go up[1] the stack to the next (deeper) frame: the function **g** was called from the function **f** in line 6 (`x = g(num, d)`).
- This function in turn was called from function **main** in line 10 (`f(5, -3, 2)`)
- And **main** function was called directly from the module at line 12 (`main()`)...
- Which it turn was called by the Python interpreter.

Exceptions do not have to cause the crash of the program, though. We can tell the interpreter what to do if they happen by putting the code which potentially can raise (throw) an exception in the **try** block and then provide the **except** block: if an exception actually occurred when executing the **try** block,[2] its execution is interrupted and the flow of control jumps directly to the **except** block: at this moment the exception is deemed "handled" and the program continues

```
a, b = 1, 0
try:
    z = a // b
    print(z)     # will not be executed
except:
    print('Exception handled')

print('Continuing...')
```

and we get

```
Exception handled
Continuing..
```

In the example above, we used just **except**, without specifying what kind of exception we want to handle here: exceptions of all types will be "caught".

It's better to be more specific, though. After a **try** block, we can have several **except** blocks for various types of exceptions: the first matching will be executed and all the remaining ones – ignored The program

```
from math import sqrt

a, b = -1, 0
try:
    x = sqrt(a)
    y = a // b
except ValueError:
    print('ValueError occurred')
except ZeroDivisionError:
    print('ZeroDivisionError occurred')
```

[1] Why *up* and not *down*? In most architectures the stack grows towards smaller addresses, so if we visualize the stack with higher addresses at the top, it grows downwards, and unwinds upwards.

[2] ...what actually can happen in a function invoked in this block, or in a function invoked by this function, etc.

```
a, b = 1, 0
try:
    x = sqrt(a)
    y = a // b
except ValueError:
    print('ValueError occurred')
except ZeroDivisionError:
    print('ZeroDivisionError occurred')
```

prints

```
ValueError occurred
ZeroDivisionError occurred
```

Of course, one has to remember about the class hierarchy: an **except** clause with a specified type will match not only this type exactly, but also its subtypes. Something like this

```
try:
    # ...
except LookupError:
    # ...
except IndexError:
    # ...
```

doesn't make sense, as **IndexError** happens to be a subtype of **LookupError**, so exceptions of this type will be caught by the first **except** clause, making the second unreachable. However

```
try:
    # ...
except IndexError:
    # ...
except LookupError:
    # ...
```

might be reasonable: the first **except** will catch **IndexError** errors, and the second **LookupError** itself and its subtypes that aren't **IndexError** errors (in this case it would be **KeyError** and, perhaps, some user defined exceptions).

However, in this way, we will not have access to the object representing the exception which may contain some useful information. Using `as name`, we can bind any name to this object and get access to its methods and attributes:

```
import traceback

a, b = 1, 0
try:
    z = a // b
except ArithmeticError as ex:
    print('*** 1. Info:\n', type(ex), str(ex))
    print('*** 2. Attributes:\n', str(dir(ex)))
    print('*** 3. Doc:\n', ex.__doc__)
    print('*** 4. Args:\n', type(ex.args), str(ex.args))
    print('*** 5. Traceback:\n', type(ex.__traceback__),
                                str(ex.__traceback__))
    print('*** 6. Trace:\n' + '-'*35)
    traceback.print_exception(ex)
    print('-'*35)
```

prints

```
*** 1. Info:
<class 'ZeroDivisionError'> integer division or modulo by zero
*** 2. Attributes:
['__cause__', '__class__', '__context__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setstate__', '__sizeof__',
 '__str__', '__subclasshook__', '__suppress_context__',
 '__traceback__', 'add_note', 'args', 'with_traceback']
*** 3. Doc:
Second argument to a division or modulo operation was zero.
*** 4. Args:
<class 'tuple'> ('integer division or modulo by zero',)
*** 5. Traceback:
<class 'traceback'> <traceback object at 0x7fa29a45ac00>
*** 6. Trace:
-----------------------------------
Traceback (most recent call last):
File "/home/werner/python/exceptions.py", line 5, in <module>
    z = a // b
        ~~^^~~

ZeroDivisionError: integer division or modulo by zero
```

Let's look at output of consecutive **print** statements in the **except** clause

*** **1.** Real type of the exception was **ZeroDivisionError**, which is a subtype of **ArithmeticError**. The **str** function prints the message taken from attribute args of the object ex (see below).

*** **2.** As we can see, the object ex is quite rich and has many attributes.

*** **3.** The __doc__ attribute contains documentation string.

*** **4.** The args attribute is a tuple of arguments passed to the constructor when ex was created; most often it has one element only — a description of the error. Contents of args will be printed by the **__str__** method, in particular when the stack trace is printed. We can define this args by passing any arguments when raising an exception, from the standard library or our own, "by hand" (see below).

*** **5.** The __traceback__ attribute contains information on the stack frames.

*** **6.** The __traceback__ attribute will be used by **print_exception** function to print the stack trace.[1]

[1] Similarly to **printStackTrace** in Java.

It is also possible to handle several unrelated types of exceptions in a single **except** clause — we just name them as a parenthesized tuple. For example,

```
from math import sqrt

a, b = -1, 0
try:
    x = sqrt(a)
    y = a // b
except (ValueError, ZeroDivisionError) as ex:
    print(type(ex).__name__, ex)
print('Exception handled')

print()

a, b = 1, 0
try:
    x = sqrt(a)
    y = a // b
except (ValueError, ZeroDivisionError) as ex:
    print(type(ex).__name__, ex)
print('Exception handled')
```

prints

```
ValueError math domain error
Exception handled

ZeroDivisionError integer division or modulo by zero
Exception handled
```

We can raise (throw) an exception "by hand" using the **raise** keyword. Its argument must be an object of an existing class representing an exception: from the standard library or our own class inheriting from **Exception**. Passing only the name of a class is equivalent to creating an object passing no arguments to its constructor. So the three following forms

```
e = Exception()
raise e

raise Exception()

raise Exception
```

are equivalent.

It is also possible to use just **raise** without specifying any type of exception. This just re-raises the exception that is currently active, what means that it can only be used inside an **except** clause. It can be useful in situations when we want to print/log an information about the error, but at the same time pass the exception upstream to the caller, who will decide what to do.

In the example below, if an error occurred, we print some information and re-raise the exception (line ❷) which, therefore, will be handled by the caller. Otherwise, we return the result of our function in the **else** clause (line ❸, see below). The caller of our function (line ❹) will print either the result or information that the invocation failed, but even then will let the program continue:

Listing 54 CAH-Reraise/Reraising.py

```
#!/usr/bin/env python

import math

def fun(a, b):
    try:
        sq = math.sqrt(a)
        lo = math.log(b)
        res = sq/lo
    except Exception as e:                              ❶
        print(f'{type(e).__name__} in fun')
        raise                                           ❷
    else:                                               ❸
        return res

args = [(4, math.e), (4, 1), (-4, 1), (4, -1), (4, math.e**2)]
for a, b in args:
    print(f'\n*** Calling fun with {a=}, {b=}')
    try:                                                ❹
        r = fun(a, b)
        print(f'Result is {r}')
    except:                                             ❺
        print('This call failed: continuing')
```

The program prints

```
*** Calling fun with a=4, b=2.718281828459045
Result is 2.0

*** Calling fun with a=4, b=1
ZeroDivisionError in fun
This call failed: continuing

*** Calling fun with a=-4, b=1
ValueError in fun
This call failed: continuing

*** Calling fun with a=4, b=-1
ValueError in fun
This call failed: continuing

*** Calling fun with a=4, b=7.3890560989306495
Result is 1.0
```

The above example illustrates also the so called “catch all” **except** clause. For example, we use it line ❶, specifying, as the type of the exception, just Exception. This will catch (almost) all exceptions, because all exceptions, directly or indirectly, inherit from Exception (there are some exception classes even higher in the hierarchy of classes describing exceptions, but those should almost never be handled). If we don't care about the object representing the exception, we just don't bind any name to it, writing just except:, as in line ❺. However, keep in mind that except: (with no type of exception specified) is not recommended, as it will also catch exceptions which should *not* be handled, like SystemExit, KeyboardInterrupt or GeneratorExit.

As in Java, after all except clauses, we can add (optionally) a finally clause with a code which will be executed always, whether an exception occurred or not. Even if try or except clauses contain the return statement, just before returning the finally clause *will* be executed.

```
>>> def f(arg):
...     try:
...         return len(arg)
...     finally:
...         print('finally executed')
...
>>> res = f('a string')
finally executed
>>> print(res)
8
>>> res = f(7.5)
finally executed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f
TypeError: object of type 'float' has no len()
```

Notice that, as in the example above, a finally block may also be used without any except blocks!

If the flow of control enters the finally block with some exception unhandled, then, as we have seen in the above example, it will be re-raised after the code of this block has been executed. However, if this code contains return (or break, or continue, if we are inside a loop), then the exception will *not* be re-raised:

```
>>> def f(arg):
...     try:
...         return len(arg)
...     finally:
...         print('finally executed')
...         return 'ERROR'
...
>>> res = f(7.5)
finally executed
>>> print(res)
ERROR
```

Unlike Java, Python supports also the so called else clause. It's rarely used, but if you use it, you have to place it after except clauses and before finally (if present). This clause will be executed *only* if there was *no* exception in the **try** clause. We can see it in the small example below

Listing 55 CAC-Clauses/Clauses.py

```python
#!/usr/bin/env python

def fun(num, denom):
    try:
        print('In try')
        z = num // denom
    except ArithmeticError as zero:
        print('Exception occurred')
        print(zero)
        return None
    else:
        print('returning result:', z)
        return z
    finally:
        print('finally executed just before returning!')

print('*** This will go smoothly')
print(fun(27, 4))
print('*** This will raise an exception')
print(fun(31, 0))
```

which prints

```
*** This will go smoothly
In try
returning result: 6
finally executed just before returning!
6
*** This will raise an exception
In try
Exception occurred
integer division or modulo by zero
finally executed just before returning!
None
```

If there is no exception in the **try** clause, we jump to **else** and return z (but notice that **finally** will be executed anyway). If there is an exception, the **except** clause will be executed: here too, just before returning **None**, the **finally** clause is executed.

Another useful feature of exceptions in Python is that we can pass any arguments to the constructor of the exception and they will be "remembered" (as a tuple) in the args attribute of the object created. In particular, args will be printed by the `__str__` method, and also when the stack trace is printed:

```
>>> ex = Exception('Raising exception just for fun', 42, '!!!')
>>> ex.args
```

```
('Raising exception just for fun', 42, '!!!')
>>> raise ex
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: ('Raising exception just for fun', 42, '!!!')
```

Most often, we just send a single string to the constructor and this string will become the message we'll see when the exception happens.[1] This mechanism works when we are creating the exception object. But it's also possible to add some information to an existing object: all exception classes have method **add_note** which takes a string and appends it to the list of "notes" kept by the object in its `__notes__` attribute. This list will appear when the stack trace is printed.

This may be useful when an exception is re-raised and traverses several stack frames before reaching an appropriate **except** clause or the crash of the program. Adding notes, we can provide some information "on the fly" to an existing exception object, as in the example below

```
def funb(a, b):
    try:
        return a // b
    except Exception as e:
        e.add_note(f'In funb with args {a=}, {b=}')
        raise

def funa(x, y):
    try:
        return funb(x+y, x-y)
    except Exception as e:
        e.add_note(f'In funa with args {x=}, {y=}')
        raise

r = funa(5, 5)
print(f'{r=}')
```

which prints

```
Traceback (most recent call last):
  File "/home/werner/python/Notes.py", line 17, in <module>
    r = funa(5, 5)
        ^^^^^^^^^^
  File "/home/werner/python/Notes.py", line 12, in funa
    return funb(x+y, x-y)
        ^^^^^^^^^^^^^^
  File "/home/werner/python/Notes.py", line 5, in funb
    return a // b
        ~~^^~~
ZeroDivisionError: integer division or modulo by zero
In funb with args a=10, b=0
In funa with args x=5, y=5
```

[1] It's conventional to use messages starting with a lowercase letter and without a period at the end.

and where information on arguments passed to functions is added to `__notes__` attribute of the exception object.

In C++/Java, exceptions are treated as necessary evil; something that should not happen in the ideal world, but that we have to accept knowing that the world is not perfect. Python, however, has more liberal approach. As we have seen, in some contexts, exceptions are used "on purpose", without any catastrophic events happening; for example, every time when an iteration stops (see sec. 2.5, p. 24 and chap. 6, p. 93).

Exception may also be used intentionally, for example, to break out of deeply nested loops. Instead of creating some Boolean variables to break across several levels of loops,[1] it's more clear and simpler to use exceptions.

In the example below, we search for three indices (in ascending order) of three elements of the list lst which would sum up to a specified value sumOf3. When we find them, there is no point to go any further, so we raise an exception (line ❶) passing the solution (current values of a, b and c) to the object representing the exception; as we know, it will become an attribute args of this object. In **except** block we just "extract" it from ex (❷). If there was no exception, the **else** block is executed (❸), setting the solution to `(-1, -1, -1)` which signals that there is no solution:

Listing 56 CAF-ExcBreak/StrangeExc.py

```
#!/usr/bin/env python

lst = [9, 1, 5, 7, 11, 10, 4]
sumOf3 = int(input('Expected sum: '))

try:
    for a in range(len(lst)-2):
        for b in range(a+1, len(lst)-1):
            for c in range(b+1, len(lst)):
                if lst[a] + lst[b] + lst[c] == sumOf3:
                    raise Exception(a, b, c)   ❶
except Exception as ex:
    res = ex.args                              ❷
else:
    res = (-1, -1, -1)                         ❸

print(res)
```

Running the program could look like this:

```
python> ./StrangeExc.py
Expected sum: 11
(-1, -1, -1)
python> ./StrangeExc.py
Expected sum: 10
(1, 2, 6)
```

[1] There are no labeled loops that we know from Java.

10.2 Built-in and custom exceptions

Standard Python provides many types of exceptions in the form of a hierarchy of classes rooted at BaseException. These exceptions may be raised by standard functions, but we can add our own types of exceptions by inheriting from one of the standard exceptions: Exception or its subclass — it's *not* recommended to inherit directly from BaseException and from its children SystemExit, KeyboardInterrupt, or GeneratorExit). Also, inheriting from multiple exception classes can lead to unpredictable behavior and should be avoided.

The list of standard exceptions can be found in the official documentation[1] and is reproduced below (this corresponds to version 3.10; more types are being added in subsequent versions of Python):

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |    +-- ModuleNotFoundError
    +-- LookupError
    |    +-- IndexError
    |    +-- KeyError
    +-- MemoryError
    +-- NameError
    |    +-- UnboundLocalError
    +-- OSError
    |    +-- BlockingIOError
    |    +-- ChildProcessError
    |    +-- ConnectionError
    |    |    +-- BrokenPipeError
    |    |    +-- ConnectionAbortedError
    |    |    +-- ConnectionRefusedError
    |    |    +-- ConnectionResetError
    |    +-- FileExistsError
    |    +-- FileNotFoundError
    |    +-- InterruptedError
    |    +-- IsADirectoryError
    |    +-- NotADirectoryError
```

[1] https://docs.python.org/3/library/exceptions.html

```
        |    +-- PermissionError
        |    +-- ProcessLookupError
        |    +-- TimeoutError
        +-- ReferenceError
        +-- RuntimeError
        |    +-- NotImplementedError
        |    +-- RecursionError
        +-- SyntaxError
        |    +-- IndentationError
        |         +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
        |    +-- UnicodeError
        |         +-- UnicodeDecodeError
        |         +-- UnicodeEncodeError
        |         +-- UnicodeTranslateError
        +-- Warning
            +-- DeprecationWarning
            +-- PendingDeprecationWarning
            +-- RuntimeWarning
            +-- SyntaxWarning
            +-- UserWarning
            +-- FutureWarning
            +-- ImportWarning
            +-- UnicodeWarning
            +-- BytesWarning
            +-- EncodingWarning
            +-- ResourceWarning
```

In most cases, it's recommended to use one of the built-in exceptions. Notice that the essential information is usually conveyed by the very name of the exception type, for example

ModuleNotFoundError or ImportError when there is a trouble with finding or loading a module;

NameError when a name is not defined;

AttributeError when a non-existent attribute is referenced;

IndexError when an out-of-range index is used on a sequence;

KeyError when a non-existent key of a dictionary is used;

TypeError when an operation or function gets an object of an inappropriate type;

ValueError when an operation or function gets an object of the right type but wrong value;

ZeroDivisionError when the second operand of division or modulo operation is 0;

UnicodeError when there is a trouble with encoding or decoding;

RecursionError when the maximum recursion depth is exceeded.

If none of the built-in exception classes suits your needs, you can create your own one. It should inherit from **Exception** or from any of its subclasses.[1] Of course, it's a "normal" class, so you can define its attributes, methods, fields as usual. However, most often we leave it as simple as possible; even the definition

```
class MyException(Exception):
    pass
```

would often be sufficient, if just its name is what we really need.

In the example below, we define a class **OutOfRange**. It is designed to be used when a certain value does not belong to a range to which it should belong. Therefore, it's natural that its parent class is **ValueError**. The constructor takes the offending value (val) and lower and upper bounds of the range to which val should belong. It just invokes the constructor of the superclass, passing this information in the form of one string:

Listing 57 CAJ-RangeError/Range.py

```
#!/usr/bin/env python

class OutOfRange(ValueError):
    def __init__(self, val, lower, upper):          ❶
        super().__init__(f'Got {val} which is out'
                         f' of [{lower}, {upper}]')


def getMonthName(n):
    names = ['January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'December']
    if (n < 1 or n > 12): raise OutOfRange(n, 1, 12)
    return names[n-1];

m = getMonthName(13)
print(m)
```

The program prints

```
  File "/home/werner/python/Range.py", line 16, in <module>
    m = getMonthName(13)
        ~~~~~~~~~~~~~~~~
  File "/home/werner/python/Range.py", line 13, in getMonthName
    if (n < 1 or n > 12): raise OutOfRange(n, 1, 12)
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^
OutOfRange: Got 13 which is out of [1, 12]
```

10.3 Assertions

We already know assertions from Java and C++. Similar mechanism is also supported by Python.

[1] See sec. ??, p. ?? about inheritance.

The assert statement is simple:

```
assert expression
```

or

```
assert expression, message
```

Here, expression is any expression whose value will be checked for “truthiness”: if it evaluates to True, nothing happens, if to False, the AssertionError exception is raised. This exception should never be handled: expression is expected to be always true, and if it isn’t, then it indicates some serious problem with the logic of our program. The optional message will be passed to the constructor of the superclass and printed when the exception occurs.

The assert statement is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

lub

```
if __debug__:
    if not expression: raise AssertionError(message)
```

But what is __debug__? This is a global variable which is initialized at the beginning of the program and cannot be modified. Normally, it is set to True. When the program is debugged and ready to be released, you can run it with __debug__ set to False, by launching the Python interpreter with optimization (option ‘-O’ – that’s ‘oh’, not zero). Then asserts are not even checked, so the program may be a little faster, as this example illustrates:

Listing 58 CAL-Assert/FiboAssert.py

```python
def fibo(n):
    assert n >= 0, 'Argumnet of fibo must be non-negtive'
    if n <= 1: return n
    return fibo(n-2) + fibo(n-1)

print('fibo(40) =', fibo(40), end=' ')
```

We measure the execution time with and without asserts enabled

```
> time python FiboAssert.py
fibo(40) = 102334155 14.312u 0.000s 0:14.31 100.0% 0+0k 0+0io 0pf+0w
> time python -O FiboAssert.py
fibo(40) = 102334155 11.949u 0.003s 0:11.95 99.9%  0+0k 0+0io 0pf+0w
```

and we see that even in this extremely contrived[1] example the difference is not substantial; we have gained 16 %.

[1] When calculating F_{40} recursively , the function fibo is called more than 300 million times, each time checking the assertion.

Chapter 11

Classes

11.1 Introduction

As in other object-oriented languages, it's possible to create our own type of data by defining a class — object of a special type that can serve as a "producer" ("factory") of objects of this new type. But of what type is this object? The answer is: all classes are themselves *objects* of type **type**!

```
>>> class AClass():
...     pass
>>> a = AClass()
>>> type(AClass)
<class 'type'>
>>> type(a)
<class '__main__.AClass'>
```

Actually objects — and, consequently, also classes — are essentially just dictionaries.

Normally, classes define a set of *attributes* (or *members*) that will be shared by all objects of this class. In Python, these are

- *Non-static methods,* i.e., functions which will always take, as their first argument, the reference to an object of the class. By convention, the corresponding parameter in the definition of a method is called self.
- *Class data members* which "belong" to the class itself, and not to objects of this class — they correspond to static fields in C++/Java.
- *Static methods* — functions which behave like 'normal' functions, but belong to the namespace of the class — they correspond to static methods in C++/Java.
- *Class methods,* i.e., functions which will always take, as their first argument, the reference to an object representing a class (not necessarily *this* class, very often it is a class derived from *this* class). By convention, the corresponding parameter in the definition of a method is called cls. There is no any corresponding construct in C++/Java.

What can be surprising here is the lack of "normal", non-static fields that would exist in every object of a class separately (as in C++/Java) — we will talk about it in a moment.
Definition of the simplest possible class in Python looks like this

```
class Klazz:
    pass
```

As definitions of functions, definition of a class is an *executable* statement: an object representing the class is created and bound to the name **Klazz**. It seems that such a class is completely "empty". Actually, it isn't:

```
>>> class Klazz:
...     pass
...
>>> dir(Klazz)
```

```
['__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__']
>>>
>>> callable(Klazz)
True
>>> k = Klazz()           # creating object
>>> type(k)
<class '__main__.Klazz'>
>>> k.__hash__()          # __hash__ already works
8776042262012
>>>
>>> AnotherName = Klazz   # new reference to the same object
>>> n = AnotherName()     # creating object
>>> type(n)
<class '__main__.Klazz'>
>>> n.__sizeof__()
32
```

Classes, as everything else in Python, are represented by objects. These objects are callable, as seen in the above snippet. Calling a class object returns an object (instance) of this class.

Objects representing classes are *not* immutable: we can, for example, add attributes to them, as well as to objects which are instances of this class.

A simple example (we will talk about the details shortly):

Listing 59 ABC-Members/Members.py

```python
#!/usr/bin/env python

class Person:
    def __init__(self, name):
        self.name = name
    def showName(self):
        print(self.name)

mary = Person("Mary")
mary.showName()         # these two invocations
Person.showName(mary)   # are equivalent

try:
    print(mary.age)
except AttributeError:
    print('Mary has no age')

mary.age = 23                                   ❶
print('And now she has one!')
print(mary.age)

Person.showAge = lambda p: print(p.age)          ❷
print('And now the class has a new method!')
mary.showAge()
```

In line ❶, we add an attribute age (which was not mentioned in the definition of the class **Person**) to the single object of the class; in line ❷, a new method was added to an existing class! The program prints

```
Mary
Mary
Mary has no age
And now she has one!
23
And now the class has a new method!
23
```

Let us now consider the example below:

Listing 60 BBB-Classes/Classes.py

```
#!/usr/bin/env python

class Klazz:
    pass

def hello(self, s):                 # just a function
    print(s, 'self =', self)

Klazz.hello = hello                 # will be attribute of Klazz

print('* 1*', hello)
print('* 2*', Klazz.hello)

k = Klazz()                         # creating object
Klazz.hello(k, '* 3* HelloK,')
k.hello('* 4* HelloK as method,') # k will be first argument


class Clazz:
    def hello(self, s):             # will be attribute of Clazz
        print(s, 'self =', self)

print('* 5*', type(Clazz.hello))

c = Clazz()
Clazz.hello(c, '* 6* HelloC,')
c.hello('* 7* HelloC as method,') # c will be first argument

```

```
29 Klazz.att = 'Attribute of Klazz'
30 Clazz.att = 'Attribute of Clazz'
31
32 cc = Clazz()
33 print('* 8*', Klazz.att)
34 print('* 9*', k.att)              # k created before
35 print('*10*', Clazz.att)
36 print('*11*', c.att)              # c created before
37 print('*12*', Clazz.__dict__)
38 print('*13*', cc.__dict__)
39
40 Clazz.hello('John', '*14* Mary,')
```

The program prints

```
* 1* <function hello at 0x7fd6aa503d90>
* 2* <function hello at 0x7fd6aa503d90>
* 3* HelloK, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 4* HelloK as method, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 5* <class 'function'>
* 6* HelloC, self = <__main__.Clazz object at 0x7fd6aa396f50>
* 7* HelloC as method, self = <__main__.Clazz object at 0x7fd6aa396f50>
* 8* Attribute of Klazz
* 9* Attribute of Klazz
*10* Attribute of Clazz
*11* Attribute of Clazz
*12* {'__module__': '__main__',
      'hello': <function Clazz.hello at 0x7fd6aa3c6440>,
      '__dict__': <attribute '__dict__' of 'Clazz' objects>,
      '__weakref__': <attribute '__weakref__' of 'Clazz' objects>,
      '__doc__': None, 'att': 'Attribute of Clazz'}
*13* {}
*14* Mary, self = John
```

We create an "empty" class Klazz and a normal two-parameter function hello. Here, the name of the first parameter, self, doesn't mean anything special. Then (line 9), we create a new attribute of the the Klazz object: the name of the attribute is hello and it refers to the object representing the hello function, what can be seen from the output of lines 11-12

```
* 1* <function hello at 0x7fd6aa503d90>
* 2* <function hello at 0x7fd6aa503d90>
```

Now, in line 14, we call the Klazz object and we get k – an instance (object) of this class.

Let's look at lines 15-16. First we call hello from Klazz object directly. We pass k as the first argument, but it could be anything else.

The second (line 16) invocation is different – here, we call the same function, but now *on* the k object (this is called *dot notation* and you know it from C++/JAVA). The interpretation is: call the function hello from the class of k (which is Klazz), and *pass* k *as the first argument.* This is what we already know from other languages, like C++/Java, although the syntax used in line 15 wouldn't work in these languages. From the output of lines 15-16

```
* 3* HelloK, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 4* HelloK as method, self = <__main__.Klazz object at 0x7fd6aa396da0>
```

we can see that in both cases the first argument of **hello** was exactly the same object. In the first case, **hello** function is treated as a normal function, the reference to which happens to be an attribute of our class. The self argument doesn't have to be of any particular type. However, the second syntax (dot notation) treats **hello** as *a method* of the class. It can be invoked only on an object of the class, and the reference to this object will be passed as the first argument (the corresponding parameter is named, although by convention only, self).[1]

In line 19, we define a class **Clazz**. This time, definition of the **hello** function is given inside the definition of the class, making it a method of the class (and its attribute). It will be called on objects of the class, and references to these object will always be passed as the first argument.

[Note: It is still possible, though, to call it as an unbound function (lines 26 and 40), passing, as the first argument, the reference to an arbitrary object, as seen in line 40 of the code. Using it in this way is confusing and should be avoided.]

In lines 29-30, we create additional attribute att in both classes. We can refer to this attribute by the class name, but also through objects of the classes created even *before* the classes were modified (lines 33-36).

In lines 37-38, we print the __dict__ attribute of both the the class itself and an object of this class.[2] This attribute is a dictionary of attributes of a class/object. As seen from the output of these lines, method **hello** and att are attributes of the class, but not of the object. Nevertheless, they will be found through the object, as attributes of classes are looked up if not found as attributes of objects.

Any class may have as many methods as we wish. But what about fields (data members)? This is different than in C++/Java. In these languages, we have to define them when defining a class and then all objects of this class will have them (their values can be, of course, different). All objects of a class have the same set of fields, the same size and the same internal structure. Not so in Python. In principle, one can create objects of a class with completely different and unrelated fields (attributes), although in practice this would lead to an unmanageable chaos.

In order to define fields that we want objects of our class to always possess, we can use the *constructor*. In Python, this is a special (magic) "dunder" method **__init__**.
Note that there can be only one constructor, as there is no function overloading in Python! As in C++/Java, the constructor will be called at the very end of the process of creating an object, when this object already exists; it will be called on this object, so the reference to it will be passed as the first argument (the corresponding parameter is traditionally called self). Inside the constructor, we can *create* attributes and assign values to them (which normally will depend on other arguments passed to the constructor). As the **__init__** function, if defined, will be called on all objects that we create, they all will have the same fields (with different values).

Note that we don't specify accessibility of member fields or methods: they are all public *par naissance*. It means that we have direct access to the fields (attributes) of objects by using, common for many languages, *dot notation:*

[1] In C++/Java this parameter is not mentioned on the list of parameters of methods, so there is no way to give it a name. Therefore, this name is once and for all fixed: this.

[2] We could have used instead the built-in function **vars**.

```
class Person:
    def __init__(self, n, y):
        self.name = n
        self.year = y

john = Person("John", 2001)
mary = Person("Mary", 2005)

print(f'{john.name}({john.year})')
print(f'{mary.name}({mary.year})')
```

prints

```
John(2001)
Mary(2005)
```

There is no name conflict between names of parameters and names of fields here. Therefore, in C++/Java, we could leave out the this reference — it would be implicitly assumed. That's because the compiler already knows the names of all fields. But in Python they have not been declared anywhere yet. Consequently, the reference self in `self.name = n` is *always* required. Generally, wherever this is assumed, but not necessary in C++/Java, the self reference must be specified explicitly in Python.

11.2 Constructing objects

We haven't talked about inheritance yet, but we remember that in C++/Java a constructor of the base class(es) is always called during creation of an object of a derived class (even before entering the constructor of the derived class). What about Python?[1] Here, the situation is different:

- If our class does *not* define the **__init__** method, the constructor of the base class *will* be invoked;
- If our class *does* define the **__init__** method, the constructor of the base class will *not* be implicitly invoked, although we can call it from the constructor of the derived class “manually” using **super**, as in the example below.[2]

Listing 61 BBD-Ctors/Ctors.py

```
#!/usr/bin/env python

class Base:
    def __init__(self):
        print('init of Base')

class DerivNoInit(Base):
    pass

```

[1] In Python, as in Java (but not C++), all classes inherit, directly or indirectly, from the class **object**.

[2] In Python, we specify the base class in round parentheses just after the name of a derived class being defined.

```
class DerivWithInit(Base):
    def __init__(self):
        print('init of DerivWithInit')

class DerivCallingBase(Base):
    def __init__(self):
        super().__init__()
        print('init of DerivCallingBase')


print('Creating DerivNoInit object')
dn = DerivNoInit()

print('\nCreating DerivWithInit object')
dw = DerivWithInit()

print('\nCreating DerivCallingBase object')
dc = DerivCallingBase()
```

The program prints

```
Creating DerivNoInit object
init of Base

Creating DerivWithInit object
init of DerivWithInit

Creating DerivCallingBase object
init of Base
init of DerivCallingBase
```

Note that invoking super in the constructor of the derived class does not have to be on the first line (as in Java); equally well, we can call it at the very end.

Creating an object is, as in C++/Java, a two-step process. First, memory must be allocated and the object has to be created. Then, when the object is already fully created, a constructor is invoked to "configure" it. Normally, the first step is executed automatically by the __new__ method of the object class, after which the __init__ is invoked on the object just created. However, we can separate these two steps, calling __new__ and __init__ "by hand":

```
class Person:
    def __init__(self, n):
        self.name = n
        print('init called')

print('Calling new')
p = object.__new__(Person)
print('dict of p:', p.__dict__)    # empty

print('Calling init')
```

```
p.__init__('John')
print('dict of p:', p.__dict__)    # now has name

print(p.name)
```

which prints

```
Calling new
dict of p: {}
Calling init
init called
dict of p: {'name': 'John'}
John
```

Note that the __new__ function takes reference to the class, object of which we are creating (the corresponding parameter is by convention called cls).

It is also possible to define our own __new__ method inside the definition of our class. Then the __new__ function from class **object** will not be called automatically, we have to do it ourselves and return the object created; on this object, the __init__ method *will* be invoked automatically:

```
class Person:
    def __new__(cls, *args, **kwargs):
        print('Creating object')
        ob = object.__new__(cls)
        return ob

    def __init__(self, name):
        print('init called')
        self.name = name

p = Person('John')
print(p.name)
```

prints

```
Creating object
init called
John
```

Note the signature of the __new__ function: its first argument is a class (reference to the object representing a class), but then `*args, **kwargs` means "whatever": normally, __new__ doesn't use these arguments — they will be passed to the __init__ constructor, as the remaining arguments (after the reference to the created object – self), as the program below demonstrates:

Listing 62 BBG-NewInit/NewInit.py

```
#!/usr/bin/env python

class AClass:
    def __new__(cls, *args, **kwargs):
        print('__new__')
        ob = object.__new__(cls)
        return ob

    def __init__(self, *args, **kwargs):
        print('__init__', end=': ')
        if len(args) == 0:
            print('No var args;', end=' ')
        else:
            print('Var args:', end=' ')
            for i, v in enumerate(args):
                print(f'{i+1} -> {v};', end=' ')
        if len(kwargs) == 0:
            print('No kwargs;')
        else:
            print('KWargs:', end=' ')
            for k, v in kwargs.items():
                print(f'{k} : {v}; ', end = ' ')
            print()

o1 = AClass()
o2 = AClass(1, 2)
o3 = AClass(1, 2, name='John', age=27)
o4 = AClass(name='John', age=27)
```

The program prints

```
__new__
__init__: No var args; No kwargs;
__new__
__init__: Var args: 1 -> 1; 2 -> 2; No kwargs;
__new__
__init__: Var args: 1 -> 1; 2 -> 2; KWargs: name : John;  age : 27;
__new__
__init__: No var args; KWargs: name : John;  age : 27;
```

As a matter of fact, if we define our own version of the __new__ function, we create an object there anyway, so we can immediately configure it using the passed arguments without even providing __init__:

```
class Person:
    def __new__(cls, name):
        ob = object.__new__(cls)
            # or: ob = super().__new__(cls)
        ob.name = name
        return ob

p = Person('John')
print(p.name)
```

what prints

```
John
```

11.3 Instance and static fields

Contrary to Java/C++, instance (belonging to individual objects) fields are *not* a part of class definition. Most often they are created inside the constructor (__init__ function), although, as we already know, it's possible to add new attributes to an already existing object:

```
>>> class AClass:
...     def __init__(self, attr1):
...         self.attr1 = attr1
...
>>> ob1 = AClass('a')
>>> ob2 = AClass(1)
>>> ob1.attr1, ob2.attr1
('a', 1)
>>> ob1.attr2 = 'b'  # ob1 has additional field
>>>
>>> ob1.__dict__
{'attr1': 'a', 'attr2': 'b'}
>>> ob2.__dict__
{'attr1': 1}
>>> ob1.attr2
'b'
>>> ob2.attr2        # but not ob2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'AClass' object has no attribute 'attr2'.
```

As we can see, objects (of any class) have special attribute __dict__ which is a dictionary containing names and values of attributes (fields) added to the object. These dictionaries may be different for different objects of the same class, although obviously it's better if all objects of the same class have the same attributes of the same type (with different values, of course).

The above example shows also another feature of instance fields of objects — in ob1 the attr1 attribute is of type **str**, while in ob2 it's an **int**. Such situation, although legal, should also be avoided.

Classes are also represented by objects, and these objects may have their own attributes, not belonging to individual objects but rather to a class itself. Such fields are known (as in Java/C++) as static fields or class attributes. They *are* a part of class definition[1] and, as in Java/C++, can be accessed applying dot notation to both the class and its objects:

```
>>> class AClass:
...     statAttr = 42
...
>>> a = AClass()
>>> AClass.statAttr
42
>>> a.statAttr
42
```

[1] Again, it's possible to add/remove static fields to classes dynamically, what is not a recommended practice, though.

Static attributes are initialized in the class definition; there is no other way, as we don't specify types in Python, so the only way to "inform" the interpreter about the type is through initialization.
As the object representing a class is mutable, we can create other static attributes dynamically. When referring to static fields, current state of the class will be used:

```
>>> class AClass:
...     statAttr1 = 1
...
>>> a = AClass()
>>> a.statAttr1
1
>>> AClass.statAttr2 = 2
>>> a.statAttr2
2
```

The example shows that the object a *does* "see" the static attribute statAttr2, although it was added to the class *after* a was created. This is understandable, as statAttr1 and statAttr1 both "belong" to the class, not to the object, but the class has been modified.
We can check what belongs to what using **vars** which displays the object's __dict__ attribute

```
>>> class AClass:
...     def __init__(self, a):
...         self.a = a
...     statAttr = 7
...
>>> a = AClass(3)
>>> vars(a)
{'a': 3}
>>> vars(AClass)
mappingproxy({'__module__': '__main__',
              '__init__': <function AClass.__init__ at 0x7fca27909b40>,
              'statAttr': 7, etc. .... })
```

As we see, class attributes can be accessed through class' name or object's name. However, if we want to modify it by assigning a new value to them, only the former will work, as a.statAttr=3 would create an additional field of the object a rather than modifying existing attribute of the class.

Let us see it in another example. We create (❶) a simple class with two attributes (static fields), a and b. In line ❷ we create also an object of the class, x.

Listing 63 BBC-Attrs/Attrs.py

```
#!/usr/bin/env python

class Klass:                 ❶
    a = 'a'
    b = 'b'

x = Klass()                  ❷
print('**1**', vars(Klass))
print('**2**', vars(x))
print('**3**', x.a, x.b)
Klass.a = 'aa'                 ❸
print('**4**', vars(x))
print('**5**', x.a, x.b)
y = Klass()
y.b = 'bb'                     ❹
print('**6**', x.a, x.b)
print('**7**', y.a, y.b)
print('**8**', vars(x))
print('**9**', vars(y))
```

In the output of the three next lines

```
**1** {'__module__': '__main__', 'a': 'a', 'b': 'b',
       '__dict__': <attribute '__dict__' of 'Klass' objects>,
       '__weakref__': <attribute '__weakref__' of 'Klass' objects>,
       '__doc__': None}
**2** {}
**3** a b
```

we see that the class does have attributes a and b, but the object x doesn't. As it doesn't have them, x.a and x.b will be found in the class object. Now, in line ❸, we modify the attribute a of the class. Since object x doesn't have "its own" a, x.a still refers to the (modified) class attribute

```
**4** {}
**5** aa b
```

In line ❹, we create another object of the class, b, and we assign the value 'bb' to y.b. There is no such attribute in y, but this is an assignment, so it will be created! Now x.a and x.b still refer to class attributes and so does y.a. But y.b is now an attribute of the object y, independent of Klass.b:

```
**6** aa b
**7** aa bb
**8** {}
**9** {'b': 'bb'}
```

11.4 Instance, static and class methods

11.4.1 Non-static methods

Instance methods of classes are defined inside class as regular functions. They have to declare at least one parameter, which, by convention, is called self and plays the same rôle as this in Java/C++ (in those languages this parameter is not mentioned in the parameter list, so the name this is fixed). The method becomes an attribute of class objects and can be referred to using dot notation applied to the class — the first argument should be the reference to an object and will be accessible in the function body as self. We can, however, use traditional syntax — call the method "on" an object (not class), and then reference to the object will be passed to the method automatically, without passing it explicitly:

```
>>> class AClass:
...     def __init__(self, at):
...         self.attr = at                          # 1
...     def info(self):
...         print('Object with attr =', self.attr) # 2
...
>>> a = AClass('hello')
>>> AClass.info(a)          # 3
Object with attr = hello
>>> a.info()                # 4
Object with attr = hello
```

In line marked with '# 3', we call the functioninfo from class AClass, passing explicitly the reference to an object of this class. In the next line, marked with '# 4', we call the same method "on" a without argument: the reference a will be passed implicitly. These two forms are equivalent in Python, although, of course, the latter is normally used (in Java/C++ only the latter would be syntactically correct).

Note that inside __init__ (line '# 1') and also in info (line '# 2'), self is obligatory: in Java/C++, we could omit this in both cases.

11.4.2 Class methods

Class methods (not to be confused with static methods!) have "special" first argument. For an instance method, it's the reference to an object on which the method is invoked (and is "received" as self). For class methods, it's the reference to a *class* (and is received by the first parameter, called, by convention, cls).

In order to define a class method, we mark it with the classmethod decorator. The first parameter will correspond to the reference to a class on which the method was invoked. As a matter of fact, we *can* invoke it on an object, but then, as for static methods in Java/C++, what counts is the type of this object (its class), not the object itself:

```
class AClass:
    @classmethod
    def fun(cls, arg):
        print(cls, 'arg =', arg)

a = AClass()
AClass.fun(5)
a.fun('Hello')
```

prints

```
<class '__main__.AClass'> arg = 5
<class '__main__.AClass'> arg = Hello
```

As we can see, in both cases the first argument is the same: reference to the class, not to any object of this class.

Class methods are often used as "factory methods" which in some sense provide additional constructors. This may be useful, because the constructor "proper" (__init__) cannot be overloaded in Python, as it can be in Java/C++.

Let us consider the following program:

Listing 64 BBJ-ClassMeth/ClassMeth.py

```
#!/usr/bin/env python

from math import pi, sqrt, sin, cos, isclose

class Point:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    @classmethod
    def from_polar(cls, r, phi):                ❶
        return cls(r*cos(phi), r*sin(phi))

    @classmethod
    def from_complex(cls, z):                   ❷
        return cls(z.real, z.imag)

    def __eq__(self, other):                    ❸
        return (isclose(self.x, other.x)
            and isclose(self.y, other.y))

aux = 1/sqrt(3)
z = complex(1, aux)

p1 = Point(1, aux)
p2 = Point.from_polar(2*aux, pi/6)
p3 = p1.from_complex(z)                         ❹

print(p1 == p2, p2 == p3, p1 == p3) # output: True True True
```

The **Point** class describes points on a plane with two coordinates. A natural constructor takes just two values which are assigned to fields x and y. We would like to have another constructor, which takes polar coordinates of a point, r and φ, but there is a problem: these are also two numbers, so how **__init__** is supposed to know whether they stand for x and y, or for r and φ? Therefore, we define a class method **from_polar** (❶) which takes (apart of cls) two numbers interpreting them as r and φ of a point, creates and returns an object of class **Point** corresponding to these polar coordinates.

Another class method, **from_complex** (❷), does the same thing, but this time based on a complex number.

In line ❸, we define a “magic” method **__eq__**, so comparisons in the last line work (more about magic methods, see sec. 11.8, p. 234).

Line ❹ demonstrates that class methods can also be called on an object, although what really matters is its type only.

When creating objects in both class methods, we used cls instead of explicitly **Point**. This is intentional. Suppose, there is a class inheriting from **Point**. Then we will be able to call this inherited method on this class (or object of this class) and the functions will return automatically object of the derived class, not of class **Point**.

11.4.3 Static methods

Static methods are marked with the **staticmethod** decorator and behave as static methods in Java/C++. Their invocation has no “hidden” arguments: all arguments correspond to explicit parameters of the function. Again, we can call them on a class, or on an object: in both cases no additional information will be passed, apart from explicitly specified arguments.

In the program below, we have two static methods in class Point. One is **heronArea**, which calculates the area of the triangle specified by three points-vertices. In order to do it, we need lengths of the sides of the triangle; to find them, we use another static method **dist**.

Listing 65 BBM-StatMeth/StatMeth.py

```python
#!/usr/bin/env python

import math

class Point:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    @staticmethod
    def heronArea(p1, p2, p3):
        a = Point.dist(p1, p2)          ❶
        b = Point.dist(p2, p3)
        c = Point.dist(p3, p1)
        s = (a + b + c)/2
        return math.sqrt(s*(s-a)*(s-b)*(s-c))

    @staticmethod
    def dist(p1, p2):
        return math.hypot(p1.x-p2.x, p1.y-p2.y)

p1 = Point(1, 1)
p2 = Point(4, 1)
p3 = Point(4, 5)

print(Point.heronArea(p1, p2, p3)) # output: 6.0
```

Note that in line ❶, we have to qualify the name of **dist** function with the name of the class (Point) — this would be unnecessary in Java/C++. In Python it’s required. We remember the LEGB rule (see sec. 5.1, p. 63): the name **dist** will be first looked for in the local scope (and not found), then in the enclosing function scope, but there is no enclosing function here, and then in the global (module) scope. But **dist** is not in the global scope — what *is* in the global scope is class Point, and one of the attributes of this class is finally our **dist**.

As we can see, we could get the same effect by defining **dist** simply as the “normal” function in the same module — it would be even simpler, because we wouldn’t have to qualify **dist** in line ❶. That's one of the reasons why static methods are seldom used in Python.

In the example below, we define a class **MyDate**, objects of which represent dates. The functions **today**, **yesterday** and **tomorrow** don't need to be static but we may want to make them static to emphasize their close relationship with our class **MyDate**:

Listing 66 ABJ-Static/Dates.py

```python
#!/usr/bin/env python

import time

class MyDate:
    def __init__(self, year, month, day):       # constructor
        self.year = year
        self.month = month
        self.day = day

    def __repr__(self):
        return f'{self.day:02}/{self.month:02}/{self.year:4}'

    @staticmethod
    def today():
        t = time.localtime()
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)

    @staticmethod
    def tomorrow():
        t = time.localtime(time.time() + 86400)
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)

    @staticmethod
    def yesterday():
        t = time.localtime(time.time() - 86400)
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)


coronation = MyDate(1953, 6, 2) # calling constructor
yesterday  = MyDate.yesterday() # static factory methods
today      = MyDate.today()     #        ./.
tomorrow   = MyDate.tomorrow()  #        ./.
print("The Queen's coronation:", coronation)
print("Yesterday:             ", yesterday)
print("Today:                 ", today)
print("Tomorrow:              ", tomorrow)
```

Program drukuje

```
The Queen's coronation: 02/06/1953
Yesterday:              20/02/2024
```

```
Today:                  21/02/2024
Tomorrow:               22/02/2024
```

11.5 Duck typing

Python supports the called 'duck typing'. The interpreter does not check the type of arguments of methods/functions — it just takes it for granted that we know what we are doing. When we call a method on an object, all what counts is whether a suitable method with a given name exists in this object's class or not. The following example illustrates this feature.

Listing 67 ABD-Duck/Duck.py

```python
#!/usr/bin/env python

class A:
    def __init__(self, data):
        self.data = data
    def getData(self):                     ❶
        return self.data

class B:
    def __init__(self, something):
        self.something = something
    def getData(self) :                    ❷
        return self.something

def fun(ob):
    return ob.getData()                    ❸

print(fun( A('Lucy') ))                    ❹
print(fun( B(28)     ))                    ❺
```

The **fun** function gets an object (here `ob`) and "blindly" calls the **getData** method on it (❸). The function doesn't care if this `ob` is of type **A** (❹) or of a completely unrelated type **B** (❺). It so happens, that both classes do have a method **getData**, so the program happily runs without any complains and prints

```
Lucy
28
```

This is not possible in C++/Java, although in C++ a similar effect van be achieved using templates.

11.6 Properties

In Python, there is no access modifiers, like **protected** or **private** in Java/C++ — everything is public. This poses some problems: all members of objects, in particular fields, can be accessed from the client code without any restrictions: they can be modified or even deleted. In languages like Java/C++, we usually make data private, and expose only methods which operate on this data in a controlled way. This is a very useful and valuable feature of these languages, and we would like to be able at least to mimic this behaviour also in Python.

One possible way is to make “a gentlemen’s agreement”, that all references with names starting with a single underscore are to be treated as protected (or private). Client code, by convention, is supposed to pretend that it doesn’t even know they exist. With this convention, we can enforce the user to access them only by methods names of which do *not* start with an underscore (so they are “true” public methods). In this way, we can have fields (attributes) protected by getters/setters, as in other languages:

Listing 68 BBT-GetSet/GetSet.py

```
#!/usr/bin/env python

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def getX(self):
        return self._x

    def setX(self, v):
          # validation in setter
        if v < 0: raise ValueError('x < 0 !!!')   ❶
        self._x = v

    def getY(self):
        return self._y

    def setY(self, v):
        self._y = v

p = Point(2, 5)
p.setX(7)
print(p.getX(), p.getY())   # output: 7 5

  # but if you are not a gentleman...
p._x = -9                                        ❷
print(p._x, p._y)           # output: -9 5
```

Setters can be used to provide some control, for example validating data (as in line ❶). Of course, we can easily circumvent the “privacy” of data, as shown in line ❷.

There is, however, a more Pythonic way to handle this kind of problems: properties. Properties allow us to create methods that behave as (“public”) attributes (fields). For example, in the class **Point** above, one can turn x and y into properties and the users will be able to access them as normal data attributes, not even knowing that they are really referring to methods of an object. The main advantage of properties is that they allow you to expose attributes as part of the public API, but we are free to change the implementation.

Let us consider the following example. Class **Square** has only one instance attribute, _a, which is “private”. We define getter (❶) and setter (❷) for this attribute (the setter implements a trivial validation). There is even a deleter (❸) — for illustration, normally it’s not needed. Then we have a getter (❹) and a setter (❺) for area. As a matter of fact, there is no such attribute; they both refer to just _a. All these methods are also “private”, as their names starts with an underscore.

Listing 69 BBP-Props/Props.py

```
#!/usr/bin/env python

from math import sqrt

class Square:
    def __init__(self, a):
        self._a = float(a)

    def _getA(self):                               ❶
        return self._a

    def _setA(self, new_a):                        ❷
        if new_a < 0:
            raise ValueError('No negative sides')
        self._a = float(new_a)

    def _delA(self):                               ❸
        del self._a

    def _getArea(self):                            ❹
        return self._a**2

    def _setArea(self, new_area):                  ❺
        if new_area < 0:
            raise ValueError('No negative areas!!!')
        self._a = sqrt(new_area)

    a    = property(_getA, _setA,                  ❻
                    _delA, "Side of the square.")
    area = property(_getArea, _setArea,            ❼
                    None, "Area of the square.")


print(type(Square.a), type(Square.area))           ❽
sq = Square(5)
print(f'Square: a={sq.a}, area={sq.area}')         ❾
sq.area = 16                                       ❿
print(f'Square: a={sq.a}, area={sq.area}')
sq.a = 7
print(f'Square: a={sq.a}, area={sq.area}')
print(f'Square\'s area doc-string: {Square.area.__doc__}')
```

So far the class has no public API. But now something interesting. We create two *public* static attributes, a (❻) and area (❼) of type **property**. From the point of view of the client, they will behave as public *non-static* (instance) data members.

As seen from the output

```
<class 'property'> <class 'property'>
Square: a=5.0, area=25.0
Square: a=4.0, area=16.0
Square: a=7.0, area=49.0
Square's area doc-string: Area of the square.
```

their type is **property** — it's a class with the constructor

```
property(fget=None, fset=None, fdel=None, doc=None)
```

and it takes a getter, setter, deleter, and doc-string for a given attribute (setter, deleter and doc-string may be left out, or **None**, which is the default). But these fields (which are called *managed attributes*) are class (static) members of **Square**, so how does it work for individual objects of the class?

We use properties always specifying an object, as in lines ❾ and ❿. The reference to this object (sq in the example) is passed as the first argument to a method of property object, and on this object the setter or getter provided in line ❻ or ❼ will be called, depending on whether the property appears on the left hand of an assignment (❿ – setter) or not (❾ – getter).[1]

Another way of creating managed attributes (properties) is by using decorators, and this is the recommended approach. We will illustrate it by rewriting the example from Listing 69.

Listing 70 BBW-PropsDec/PropsDec.py

```
#!/usr/bin/env python

from math import sqrt

class Square:
    def __init__(self, a):
        self._a = float(a)

    @property                                        ❶
    def a(self):
        """Side of the square."""
        return self._a

    @a.setter                                        ❷
    def a(self, new_a):
        if new_a < 0:
            raise ValueError('No neg. sides')
        self._a = float(new_a)

    @a.deleter                                       ❸
```

[1] Or as the argument of **del** operator.

```
    def a(self):
        del self._a

    @property
    def area(self):                          ❹
        """Area of the square."""
        return self._a**2

    @area.setter
    def area(self, new_area):                ❺
        if new_area < 0:
            raise ValueError('No neg. areas!!!')
        self._a = sqrt(new_area)

print(type(Square.a), type(Square.area))
sq = Square(5)
print(f'Square: a={sq.a}, area={sq.area}')
sq.area = 16
print(f'Square: a={sq.a}, area={sq.area}')
sq.a = 7
print(f'Square: a={sq.a}, area={sq.area}')
print(f'Square\'s area doc-string: {Square.area.__doc__}')
```

We create a property named a in line ❶. Note that a will be the name of a *property*, not of a function. The implementation must correspond to the getter, and the property's doc-string will be that specified here. After that, we already have the property a with getter only, so it corresponds to a read-only attribute. Suppose, we would also like to have a setter and a deleter.[1]

In order to add a setter to the already existing property a, we decorate it with **@a.setter** (❷), where a is the property's name. Note that the name after **def** must be again exactly the same as the name of the property itself, in our case a. And the same applies to the deleter (❸).

In a similar way, we define the area property, this time without a deleter – ❹ and ❺.

The output of the program is exactly the same as that from Listing 69:

```
<class 'property'> <class 'property'>
Square: a=5.0, area=25.0
Square: a=4.0, area=16.0
Square: a=7.0, area=49.0
Square's area doc-string: Area of the square.
```

11.7 String representation of objects

Very often we want to print (or more generally, to get a string representation of) objects of our class. We handle this problem by overriding the **toString** method in Java or **operator<<** in C++. In Python, there are even two such (dunder) methods that you can define: **__str__** and **__repr__**. As is the case for other dunder methods, you almost never call them "by name". If the object you want to get a string

[1] But first ask yourself whether you really need a setter and/or a deleter…

representation of is ob, str(ob) will automatically call the __str__ method on ob, while repr(ob) will invoke __repr__.

Actually, you can call these functions without defining any methods in our class

```
class Person:
    def __init__(self, name, year):
        self.name = name
        self.year = year

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

and you will get

```
<__main__.Person object at 0x7f8ba1e07a00>
<__main__.Person object at 0x7f8ba1e07a00>
```

This is the default string representation of objects (inherited from class **object**) — it's rarely useful, similarly to the default implementation of **toString** in Java. To get something more meaningful, you can define one or both of __str__ and __repr__ methods. If you define __repr__ only, it will be used for str(ob) too:

```
class Person:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def __repr__(self):
        return 'from repr'

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

prints

```
from repr
from repr
```

However, if you define __str__ only

```
class Person:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def __str__(self):
        return 'from str'

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

then

```
from str
<__main__.Person object at 0x7f1242a13a90>
```

will be printed, so it will *not* be used for `repr(ob)`. It follows that for your own classes, you should rather define the **__repr__** method, and if another representation would be better for invocations of **str**, define additionally the **__str__** method.

The idea of having two such functions is to differentiate between the forms of the output depending on the addresee: **__str__** is meant for the human user – it doesn't have to be overly precise, but rather easy to read and understand. The **__repr__** form is more precise and intended for the developer, useful when debugging. If at all possible, it should return a string which can be directly used to reproduce the object:

```python
class Person:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def __repr__(self):
        return f'Person("{self.name}", {self.year})'

john = Person("John", 2001)

print('*1*', john)      # repr will be used

p = eval(repr(john))
print('*2*', type(p))
print('*3*', p)
```

prints

```
*1* Person("John", 2001)
*2* <class '__main__.Person'>
*3* Person("John", 2001)
```

(We used here the built-in **eval** function which evaluates its argument as a Python code.)

Objects are also converted to strings when being formatted. For example, **print** uses **__str__** as does `{object}` in f-strings. If rather **__repr__** is to be used in an f-string, we can enforce it by appending the conversion flag 'r': `{object!r}`. The form with equal sign, `{object=}`, behaves the other way around: by default **__repr__** will be used, and to get **__str__**, we have to use `{object=!s}`:

```python
class Person:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def __repr__(self):
        return f'Person("{self.name}", {self.year})'

    def __str__(self):
        return f'{self.name}({self.year})'

john = Person("John", 2001)

print('*1*', john)
print('*2*', repr(john))
print('*3*', f'{john}')
print('*4*', f'{john!r}')
print('*5*', f'{john=}')
print('*6*', f'{john=!s}')
```

prints

```
*1* John(2001)
*2* Person("John", 2001)
*3* John(2001)
*4* Person("John", 2001)
*5* john=Person("John", 2001)
*6* john=John(2001)
```

11.8 Magic methods

There are many so-called "magic" methods in Python. Their names start and end with double underscore, so they are colloquially referred to as *dunder methods*. They are special, because normally we don't call them explicitly — they will be called automatically in some contexts, for example if there is a binary operator between two objects, or a unary operator in front of an object, or by some built-in functions (**iter**, **next**, **len**, and many others), or when indexing is applied to an object, and so on.

We have already encountered some of these special methods: **__iter__**, **__next__**, **__getitem__** (sec. 2.5, p.24), **str** and **__repr__** (sec. 11.7, p.231), **new** and **__init__** (sec. 11.2, p.216), and a few others.

11.8.1 Operator overloading

As we remember, in C++, we can overload operators by defining special methods (or free functions) called **operator**+, **operator**− and so on. In Python, to get a similar effect, we define "magic" methods (but *not* free functions) with names such as **__add__**, **__sub__**, **__mul__**, etc. As in C++, there are some limitations

- We cannot overload operators for the built-in types.
- We cannot create new symbols for operators; only existing operators can be overloaded.
- Some operators cannot be overloaded: **is**, **and**, **or** and **not**.

Let us begin with arithmetic operators.

Suppose we have two object, `a` and `b`, of a class **AClass** and we want the expression `a+b` to return another object of this class which is, in some sense, the sum of the two operands.[1] All we have to do is to define an **__add__** method in our class. Then, `a+b`

[1] As a matter of fact, we can return whatever we wish, but this is what the user would expect by analogy to other types for which addition is defined; it's better not to violate the *principle of least astonishment*.

will be equivalent to `a.__add__(b)` or, more precisely, to `type(a).__add__(a, b)`. Note, that the method **`__add__`** (and similarly for other binary operators) will always be called on the *left* operand.

Similarly, for unary operators. Now there is only one operand, and the special method will be called on it without any arguments. For example, operator of bitwise negation, ∼, can be overloaded by **`__invert__`** method, and then `~a` will be equivalent to `a.__invert__()` or `type(a).__invert__(a)`.

Let's look at an example:

Listing 71 BBZ-Over/Resistor.py

```
#!/usr/bin/env python

class Resistor:
    """Describes resistor."""
    def __init__(self, r = 0):
        self.res = r                                        ❶

    @property
    def res(self):
        """Resistance of the resistor."""
        return self._res

    @res.setter
    def res(self, r):
        if r < 0: raise ValueError('Negative resistance')
        self._res = float(r)                                ❷

    def __add__(self, other):                               ❸
        return Resistor(self.res + other.res)

    def __mul__(self, other):                               ❹
        return Resistor(self.res*other.res/(self.res+other.res))

    def __repr__(self):                                     ❺
        return f'Resistor({self.res:.2f})'


r1, r2 = Resistor(), Resistor(6)
r1.res = 3
print(r1+r2, r1*r2)                                         ❻
```

The class **Resistor** describes resistors. It has one private field, `_res`, and property `res` corresponding to the resistance of the resistor. Note that in the constructor, we don't define `self._res` — we just use the property `res` (❶) which will call the setter ❷. This is because the setter could, in principle, contain some form of logging or validation (in our example, the setter ensures that `_res` is positive and of type **float**).

In line ❸, we define addition of two resistors in such a way that the result corresponds to a new resistor with its resistance equal to the total resistance of two resistors connected in series. The "multiplication" of resistors (❹) will also correspond to the total resistance of the two resistors, but for parallel connection. Additionally, we define __repr__ method (❺) in order to provide a string representation of our resistors. All these three special methods of the class will be used in line ❻. The output of the program is

```
Resistor(9.00) Resistor(2.00)
```

Let's summarize operators of this kind:

Table 4: Overloading numeric operators

Operation	Operator	Method
Addition	a + b	a.__add__(b)
Subtraction	a - b	a.__sub__(b)
Multiplication	a * b	a.__mul__(b)
Division	a / b	a.__truediv__(b)
Floor division	a // b	a.__floordiv__(b)
Remainder	a % b	a.__mod__(b)
Power	a ** b	a.__pow__(b)
Bitwise left shift	a << b	a.__lshift__(b)
Bitwise right shift	a >> b	a.__rshift__(b)
Bitwise AND	a & b	a.__and__(b)
Bitwise OR	a | b	a.__or__(b)
Bitwise XOR	a ^ b	a.__xor__(b)
Bitwise NOT	~a	a.__not__()
Unary PLUS	+a	a.__pos__()
Unary MINUS	-a	a.__neg__()

Most of these operators have the form of augmented assignments (like a += b). They also can be overloaded by defining appropriate dunder methods (just add 'i' to the name of the corresponding method). These methods should, if possible, modify self and return it:

Table 5: Overloading augmented assignments

Assignment with...	Operator	Method
addition	a += b	a.__iadd__(b)
subtraction	a -= b	a.__isub__(b)
multiplication	a *= b	a.__imul__(b)
division	a /= b	a.__itruediv__(b)
floor division	a //= b	a.__ifloordiv__(b)

Table 5: (continued)

Assignment with...	Operator	Method
remainder	`a %= b`	`a.__imod__(b)`
power	`a **= b`	`a.__ipow__(b)`
bitwise left shift	`a <<= b`	`a.__ilshift__(b)`
bitwise right shift	`a >>= b`	`a.__irshift__(b)`
bitwise AND	`a &= b`	`a.__iand__(b)`
bitwise OR	`a |= b`	`a.__ior__(b)`
bitwise XOR	`a ^= b`	`a.__ixor__(b)`

Note that we have a problem with operations which "by nature" should be symmetric. Let us define a class **Modulo7** which represents just a number, but with arithmetic operations performed modulo 7 (❶). We demonstrate addition only, but other operators could have been implemented in a similar way. We thus define the **__add__** method (❸) which adds to a given object of the class a number in an obvious way. The expression `n4 + 5` will be converted to `n4.__add__(5)` and we'll get an object corresponding to 2 (as 9 mod 7 is 2).

Listing 72 BDB-Reflect/ReflectBad.py

```python
#!/usr/bin/env python

class Modulo7:
    modulus = 7                                 ❶
    def __init__(self, num):
        self.num = num % Modulo7.modulus

    def __add__(self, val):                     ❷
        return Modulo7(self.num + val)

    def __repr__(self):
        return f'Modulo7({self.num})'


n4 = Modulo7(4)
n9 = n4 + 5                                     ❸
print(n4, n9)
n8 = 4 + n4                                     ❹
print(n8)
```

However, the program crashes, because **__add__** is called on the left operand, which is an object of our class in line ❸, but not in line ❹, where it's on the right-hand side — on the left, we have just an integer:[1]

```
Modulo7(4) Modulo7(2)
```

[1] In C++ we could define **operator**+ as a free function to handle this case, but in Python it must be an instance method.

```
Traceback (most recent call last):
  File "/home/werner/python/ReflectBad.py", line 18, in <module>
    n8 = 4 + n4                                ##+4
TypeError: unsupported operand type(s) for +: 'int' and 'Modulo7'
```

However, the symmetry can be saved. Suppose we have an operator @ corresponding to the dunder method __oper__ in a class AClass. Let a be an object of the class. Then, as we know, a @ b will be converted to a.__oper(b). But what, if a is on the right, b @ a? Then, if

- the left operand does not support the corresponding operation (or implementation returns NotImplemented),
- and is of different type than a,
- and AClass has the dunder method __roper__ (note the letter 'r')

then *reflected* operation will take place: b @ a will be converted to a.__roper__(b).

Therefore, in our example, we will add the __radd__ method (❶) which will handle the reversed order of arguments (❷)

Listing 73 BDB-Reflect/ReflectGood.py

```python
#!/usr/bin/env python

class Modulo7:
    modulus = 7
    def __init__(self, num):
        self.num = num % Modulo7.modulus

    def __add__(self, val):
        return Modulo7(self.num + val)

    def __radd__(self, other):                    ❶
       return Modulo7(self.num + other)

    def __repr__(self):
        return f'Modulo7({self.num})'


n4 = Modulo7(4)
n9 = n4 + 5
print(n4, n9)
n8 = 4 + n4                                       ❷
print(n8)
```

and now 4+n4 in line ❷ will work as expected

```
Modulo7(4) Modulo7(2)
Modulo7(1)
```

and print

```
    Modulo7(4) Modulo7(2)
    Modulo7(1)
```

The example above used addition, but, of course, the same mechanism can be applied to other operators:

Table 6: Reflected operators

Operation	Operator	Method
addition	`b + a`	`a.__radd__(b)`
subtraction	`b - a`	`a.__rsub__(b)`
multiplication	`b * a`	`a.__rmul__(b)`
division	`b / a`	`a.__rtruediv__(b)`
floor division	`b // a`	`a.__rfloordiv__(b)`
remainder	`b % a`	`a.__rmod__(b)`
power	`b ** a`	`a.__rpow__(b)`
bitwise left shift	`b << a`	`a.__rlshift__(b)`
bitwise right shift	`b >> a`	`a.__rrshift__(b)`
bitwise AND	`b & a`	`a.__rand__(b)`
bitwise OR	`b | a`	`a.__ror__(b)`
bitwise XOR	`b ^ a`	`a.__rxor__(b)`

Of particular importance are six "rich" comparison operators: ==, !=, <, <=, >, >=. They also correspond to magic methods, so, for example, `a < b` is equivalent to `a.__lt__(b)`.

Table 7: Comparison operators

Comparison	Operator	Method
less than	`a < b`	`a.__lt__(b)`
less or equal	`a <= b`	`a.__le__(b)`
grater than	`a > b`	`a.__gt__(b)`
greater or equal	`a >= b`	`a.__ge__(b)`
equal	`a == b`	`a.__eq__(b)`
non-equal	`a != b`	`a.__ne__(b)`

We can define these methods, if we want objects of our class to be comparable. For example, in the program below, we have a simple class AClass which has one instance field string of type str. Suppose we want to be able to compare its objects based on this field, according to the rules:

- Shorter strings are considered "smaller" than longer;
- For equal lengths, "smaller" string is the one earlier lexicographically, but ignoring the case.

We can define all six methods of rich comparison (note that the implementation of many of them just uses those already implemented):

Listing 74 BDE-Cmp/Cmp.py

```python
#!/usr/bin/env python

class AClass:
    def __init__(self, string):
        self.string = string

    def __lt__(self, other):
        if len(self.string) != len(other.string):
            return len(self.string) < len(other.string)
        else:
            return self.string.lower() < other.string.lower()

    def __gt__(self, other):
        return other < self

    def __ge__(self, other):
        return not (self < other)

    def __le__(self, other):
        return not (other < self)

    def __eq__(self, other):
        return self.string.lower() == other.string.lower()

    def __ne__(self, other):
        return not (self == other)

    def __repr__(self):
        return f"{self.string}"


def info(a, b):
    print(f'a<b:{a<b!s:<5}, a<=b:{a<=b!s:5}, a==b:{a==b!s:5} '
          f'a>=b:{a>=b!s:5}, a>b:{a>b!s:5}, a!=b:{a!=b!s:5}')

a, b, c, d, e, f = (AClass('Abc'),  AClass('aBC'),
                    AClass('Abcd'), AClass('z'),
                    AClass('XY'),   AClass('bcda'))
info(a, b); info(c, d); info(b, e)
lst = [a, b, c, d, e, f]
print(lst)
lst.sort()
print(lst)
```

The program prints

```
a<b:False, a<=b:True , a==b:True  a>=b:True , a>b:False, a!=b:False
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
```

```
[Abc, aBC, Abcd, z, XY, bcda]
[z, XY, Abc, aBC, Abcd, bcda]
```

what's what we would expect. However, there is a simpler way. There is a *class* decorator **total_ordering** in the ***functools*** module. Applied to a class, it will create four of the comparison methods automatically; we only have to provide two of them: **__eq__** method and *one* of **__lt__**, **__le__**, **__gt__** and **__ge__**.

In this way, the **AClass** class from the previous example can be written much more compactly:

Listing 75 BDE-Cmp/CmpDec.py

```python
#!/usr/bin/env python

import functools

@functools.total_ordering
class AClass:
    def __init__(self, string):
        self.string = string

    def __eq__(self, other):
        return self.string.lower() == other.string.lower()

    def __lt__(self, other):
        if len(self.string) != len(other.string):
            return len(self.string) < len(other.string)
        else:
            return self.string.lower() < other.string.lower()

    def __repr__(self):
        return f"{self.string}"

def info(a, b):
    print(f'a<b:{a<b!s:<5}, a<=b:{a<=b!s:5}, a==b:{a==b!s:5} '
          f'a>=b:{a>=b!s:5}, a>b:{a>b!s:5}, a!=b:{a!=b!s:5}')


a, b, c, d, e, f = (AClass('Abc'),  AClass('aBC'),
                    AClass('Abcd'), AClass('z'),
                    AClass('XY'),   AClass('bcda'))
info(a, b); info(c, d); info(b, e)
lst = [a, b, c, d, e, f]
print(lst)
lst.sort()
print(lst)
```

and gives the same output as before

```
a<b:False, a<=b:True , a==b:True  a>=b:True , a>b:False, a!=b:False
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
```

```
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
[Abc, aBC, Abcd, z, XY, bcda]
[z, XY, Abc, aBC, Abcd, bcda]
```

It should be noted here that the methods created by the total_ordering decorator might *not* be optimal; if the performance of our program is important, then it's probably better to carefully implement all six comparison methods "by hand".

11.8.2 Other magic methods

Magic methods are also used to make object of custom classes "respond" to built-in functions, or generally behave in a sensible way in various context. An example is the __str__ method which is used by str function to provide a textual representation of an object, or __len__ for the object to respond to the built-in len function. We have already encountered some of these functions, but there are many others.

Let us mention some of them.

- `__new__` and `__init__` — we already know them from sec. 11.2, p. 216.

- `__del__(self)` — will be called when the object is about to be destroyed. Note that the built-in function del decrements reference counter, and __del__ will only be called when the counter reaches 0. In most cases there is no reason to define it.

- `__str__(self)`, `__repr__(self)` and `__bytes__(self)` — the first two were covered in sec. 11.7, p. 231. The last one is called by the built-in bytes function and is supposed to return the representation of the object as a bytes object.

- `__format__(self, spec)` — is called by the built-in format function and in some other contexts. Normally, we don't define it.

- `__hash__(self)` — is called by the built-in hash function. Should return an integer. If we plan to use object of a class as elements of sets or keys of dictionaries, we should define *both* __eq__ and __hash__ consistently, i.e., if for two objects __eq__ (or ==) returns True, then their hash codes should be identical. In the example below, both __eq__ (❶) and __hash__ (❷) depend on two fields, fname and lname, guaranteeing that for objects which are deemed equal their hash code will be calculated from identical values. When implementing __hash__, it's convenient to rely on built-in hash function for objects which already have their hashes well defined, as in line ❸, where hash calculates the hash code of a tuple composed of fname and lname. Remember also, that implementation of hash is such, that the result (and, consequently, the order of elements returned when iterating over sets) may be different in every run of the program, even on the same computer in two subsequent runs.

Listing 76 BDH-Hash/Hash.py

```python
#!/usr/bin/env python

class A:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __eq__(self, other):                              ❶
        return (self.fname,  self.lname ==
                other.fname, other.lname)

    def __hash__(self):                                   ❷
        return hash( (self.fname, self.lname) )           ❸

    def __repr__(self):
        return f'A("{self.fname}","{self.lname}")'


p1, p2, p3 = (A('Don', 'Ash'), A('Sue', 'Doe'),           ❹
                               A('Ada', 'Moo'))

d  = {A('Don', 'Ash'): 2000, A('Sue', 'Doe'): 1997,       ❺
                             A('Ada', 'Moo'): 2005}

print(f'{p1}->{d[p1]} ', f'{p2}->{d[p2]} ', f'{p3}->{d[p3]}')
```

The program prints:

```
A("Don","Ash")->2000  A("Sue","Doe")->1997  A("Ada","Moo")->2005
```

Note that references p1, p2 and p3 refer to objects created in line ❹; however objects created in line ❺ as keys in dictionary d are distinct objects, with different addresses, although they point to objects representing the same persons as those referred by p1, p2 and p3,because they have the same first and last names. This works as our implementation of __hash__ and __equal__ is based on names, and not addresses, as it is in the default implementation in class **object**. Try commenting out either or both of these methods and you will see that the program crashes.

- __bool__(self), __int__(self), __complex__(self) and __float__(self) — are used when conversion to these types is needed, in particular when the object is the argument of the built-in functions **bool**, **int**, **complex**, **float**. For example, let's have a look at the following program:

Listing 77 BDN-Bool/Bool.py

```
#!/usr/bin/env python

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f'Person(name={self.name},age={self.age})'

    def __bool__(self):            # conversion to bool
        return self.age >= 18

```

```

lst = [Person("A", 13), Person("B", 19),
       Person("C", 27), Person("D", 10)]
adults = [p for p in lst if p]    ❶
print(adults) # [Person(name=B,age=19), Person(name=C,age=27)]
```

In the expression on line ❶, `[p for p in lst if p]`, the last `p` is under **if**, so it has to be converted to **bool**. Normally, anything which is not zero, not **None** and not an empty collection is deemed to be **True**. However, in class **Person**, we have overwritten the **__bool__** method, and therefore it will be used yielding **True** only for adults (with age not smaller than 18)

```
[Person(name=B,age=19), Person(name=C,age=27)]
```

● `__call__(self)`, — implementing **__call__**, we can make objects of a class to be *callable*, i.e., to behave as functions (like in C++, where we implement the **operator()** method). "Calling" an object, `obj(args)`, is equivalent to `type(obj).__call__(obj, args)` or `obj.__call__(args)`. The mechanism may be useful when we want to have something behaving like a function, but with state than can be modified and "remembered" between invocations.

For example, in the following program

Listing 78 BDR-CallAve/Ave.py

```
#!/usr/bin/env python

class Average:
    def __init__(self):
        self.sum = 0
        self.num = 0

    def __call__(self, val):
        self.sum += val
        self.num += 1
        return self.sum / self.num

av1 = Average()
av2 = Average()

for n in range(1, 11):
    print(f'av1: adding {n:2d} -> {av1(n):5.2f}', ' ',    ❶
          f'av2: adding {n**2:3d} -> {av2(n*n):5.2f}')    ❷
```

the **Average** class has two fields which correspond to the sum of numbers and the number of these numbers. Each invocation of the **__call__** method increments the field **sum** by the value passed as the argument, increments field **num** by one, and returns the new value of the arithmetic average of all numbers passed to the function so far. In lines ❶ and ❷, we pass numbers from the $[1, 10]$ interval and their squares, respectively, as arguments to two objects, **av1** and **av2** of our class. The program prints

```
av1: adding  1 ->  1.00   av2: adding   1 ->  1.00
av1: adding  2 ->  1.50   av2: adding   4 ->  2.50
av1: adding  3 ->  2.00   av2: adding   9 ->  4.67
av1: adding  4 ->  2.50   av2: adding  16 ->  7.50
av1: adding  5 ->  3.00   av2: adding  25 -> 11.00
av1: adding  6 ->  3.50   av2: adding  36 -> 15.17
av1: adding  7 ->  4.00   av2: adding  49 -> 20.00
av1: adding  8 ->  4.50   av2: adding  64 -> 25.50
av1: adding  9 ->  5.00   av2: adding  81 -> 31.67
av1: adding 10 ->  5.50   av2: adding 100 -> 38.50
```

Let us now consider the next example: here we calculate Fibonacci numbers by the recursive function **fiboNoCache** and using callable object of the **FiboCached**. In the latter case, we keep all values of the Fibonacci numbers calculated so far in a dictionary, to avoid recalculating them over and over again.

Listing 79 BDU-CallCache/Cache.py

```python
#!/usr/bin/env python

class Average:
    def __init__(self):
        self.sum = 0
        self.num = 0

    def __call__(self, val):
        self.sum += val
        self.num += 1
        return self.sum / self.num

av1 = Average()
av2 = Average()

for n in range(1, 11):
    print(f'av1: adding {n:2d} -> {av1(n):5.2f}', ' ',   ❶
          f'av2: adding {n**2:3d} -> {av2(n*n):5.2f}')   ❷
```

As we can see from the output

```
102334155 1.517100099590607e-05 sec
102334155 12.199706612002046 sec
ratio: 804146
```

execution times differ by six orders of magnitude!

There are other “dunder” methods which are sometimes useful – we will talk about them when they are needed.

Listings

List of listings

1 AAB-First/First.py . . . 11
2 AAC-Refs/Refs.py . . . 13
3 AAY-Funcs/funcs.py . . . 14
4 AAM-Pass/Pass.py . . . 21
5 AAG-Iffs/Iffs.py . . . 22
6 AAJ-While/While.py . . . 23
7 AAK-For/For.py . . . 24
8 AAN-DictLoop/DictLoop.py . . . 25
9 AAL-ForEnum/Enum.py . . . 26
10 AAO-EnumStart/EnumStart.py . . . 26
11 AAQ-MatchBasic/MatchBasic.py . . . 27
12 AAT-MatchLen/MatchLen.py . . . 28
13 AAV-MatchVS/MatchValStr.py . . . 28
14 AAX-MatchSN/MatchSeqNam.py . . . 29
15 ABB-Cond/Cond.py . . . 61
16 AAP-FunCompos/Compos.py . . . 64
17 AAR-FunCount/Counter.py . . . 65
18 AAS-Scopes/Scopes.py . . . 67
19 AAU-DefArgs/DefArgs.py . . . 70
20 AAW-ArgsKwargs/ArgsKwargs.py . . . 72
21 AAZ-Filter/Filter.py . . . 79
22 ABS-Deco/Deco.py . . . 88
23 ABT-DecoNest/DecoNest.py . . . 89
24 ABU-DecoParam/DecoParam.py . . . 90
25 ABY-DecoTime/DecoTime.py . . . 91
26 ABA-GenFib/GenFib.py . . . 103
27 ABL-UniqueWords/UniqueWords.py . . . 104
28 ABR-GenOfGen/GenOfGen.py . . . 104
29 ABO-TwoWayGener/TwoWayGener.py . . . 106
30 ABP-Chooser/Chooser.py . . . 106
31 ABV-Priming/Priming.py . . . 108
32 ABW-PushPipe/PushPipe.py . . . 109
33 ABX-PullPipe/PullPipe.py . . . 112
34 ABM-Islice/Islice.py . . . 115
35 ABN-Groups/Groups.py . . . 116
36 ACC-DefDict/DefDict.py . . . 145
37 ACE-Purchases/Purchases.py . . . 146
38 REA-Split/Splitting.py . . . 169
39 REB-Groups/Groups.py . . . 171
40 REC-Nesting/Nesting.py . . . 173
41 RED-NoGroup/NoGroup.py . . . 174
42 REE-NamedGroups/NamedGroups.py . . . 175
43 REF-BackRefs/BackRefs.py . . . 176
44 REG-Replace/Replace.py . . . 177
45 REH-Look/Look.py . . . 178
46 RER-PassCheck/PassCheck.py . . . 179
47 REI-YesNo/YesNo.py . . . 180

48 REK-RegFlags/RegFlags.py . . . 182
49 REL-FlagM/FlagM.py . . . 183
50 REM-Verbose/Verbose.py . . . 185
51 REJ-SubFun/SubFun.py . . . 190
52 REN-Matcher/Matcher.py . . . 192
53 REO-PattAttr/PattAttr.py . . . 195
54 CAH-Reraise/Reraising.py . . . 202
55 CAC-Clauses/Clauses.py . . . 204
56 CAF-ExcBreak/StrangeExc.py . . . 206
57 CAJ-RangeError/Range.py . . . 209
58 CAL-Assert/FiboAssert.py . . . 210
59 ABC-Members/Members.py . . . 212
60 BBB-Classes/Classes.py . . . 213
61 BBD-Ctors/Ctors.py . . . 216
62 BBG-NewInit/NewInit.py . . . 218
63 BBC-Attrs/Attrs.py . . . 221
64 BBJ-ClassMeth/ClassMeth.py . . . 224
65 BBM-StatMeth/StatMeth.py . . . 225
66 ABJ-Static/Dates.py . . . 226
67 ABD-Duck/Duck.py . . . 227
68 BBT-GetSet/GetSet.py . . . 228
69 BBP-Props/Props.py . . . 229
70 BBW-PropsDec/PropsDec.py . . . 230
71 BBZ-Over/Resistor.py . . . 235
72 BDB-Reflect/ReflectBad.py . . . 237
73 BDB-Reflect/ReflectGood.py . . . 238
74 BDE-Cmp/Cmp.py . . . 240
75 BDE-Cmp/CmpDec.py . . . 241
76 BDH-Hash/Hash.py . . . 242
77 BDN-Bool/Bool.py . . . 243
78 BDR-CallAve/Ave.py . . . 244
79 BDU-CallCache/Cache.py . . . 245

Index

** (operator), 12
_asdict (function), 144
_fields (function), 144
_replace (function), 144

ABC (language), 1
accumulate (function), 78, 82
accumulator, 81
__add__ (function), 236
add (function), 131
add_note (function), 205
all (function), 96
alternative (logical), 36
__and__ (function), 236
and (operator), 36, 59
annotation, 86
append (function), 123, 150
appendleft (function), 150
argument
  default, 70
arithmetic comparison operator, 57
arithmetic operator, 50
ASCII, 184
assignment, 40, 41
  chained, 40
attribute, 211, 220
attribute of pattern object, 195
await expression, 49

backreference, 176
BaseException (class), 207
bin (function), 34, 54
bit_length (function), 11
bitwise and (operator), 56
bitwise not (operator), 56
bitwise operator, 53
bitwise or (operator), 56
bitwise xor (operator), 56
__bool__ (function), 243
bool (class), 60
bool (type), 35
boolean operator, 59
break (instruction), 23
bytearray (class), 127
__bytes__ (function), 242
bytes (class), 125

C/C++, 1
__call__ (function), 244
__call__ method, 49
call (operator), 48
callable (function), 49
capitalize (function), 160
capturing group, 171
center (function), 160
chained assignments, 40
class, 211
  bool, 60
  bytearray, 127
  bytes, 125
  Counter, 146
  data member, 211
  defaultdict, 145
  deque, 150
  dict, 137
  frozenset, 130
  GeneratorExit, 207
  KeyboardInterrupt, 207
  set, 130
  SystemExit, 207
  type, 211
class attribute, 220
class decorator, 241
class method, 211, 223
clear (function), 136, 140, 152
close (function), 109
closure, 67, 76
cls, 211
cmp_to_key (function), 85
Collatz sequence, 61
collection, 118
  sequential, 118
comparison operator, 57
compile (function), 170, 186
__complex__ (function), 243
complex (type), 7, 37
comprehension, 93
  dictionary, 98
  generator, 102
  list, 94
  set, 97
conditional expression, 22, 60
conjunction (logical), 36
constructor, 215
Container (protocol), 118

continue (instruction), 23
copy (function), 48, 124, 140, 152
coroutine, 105
    pipe, 109
count (function), 115, 162
count (funkcja), 152
Counter (class), 146
CWI, 1

data member, 211
data type, 31
Decimal, 31
decorator, 85
    nested, 89
deepcopy (function), 13, 48
default argument, 70
defaultdict (class), 145
del, 123
__del__ (function), 242
deque (class), 150
dict (class), 137
dict (type), 9
dictionary, 137
dictionary comprehension, 98
difference (function), 134
difference_update (function), 136
dir (function), 18
discard (function), 131
division (operator), 51
dot notation, 215
DOTALL, 184
dropwhile (function), 113
duck typing, 15, 63, 227
dunder method, 49
dunder methods, 234
dynamic typing, 1

elements (function), 148
ellipsis type, 32
else clause, 203
end (function), 172
end (funkcja), 194
endswith (function), 162
enumerate (function), 25
enumerate object, 25
__eq__ (function), 239
escape (function), 191
except clause, 198
exception, 197
    StopIteration, 24, 93, 100
exponentiation (operator), 12, 49

expression
    await, 49
    conditional, 22, 60
    if-else, 22
    lambda, 74
    parenthesized, 42
extend (function), 123, 151
extendleft (function), 151

False, 35
falsy, 35
filter (function), 78, 79
finally clause, 203
find (function), 45, 162
findall (function), 189
finditer (function), 175, 178, 190
__float__ (function), 243
float (type), 7, 36
floor division, 51
__floordiv__ (function), 236
flow control statement, 21
for loop, 24
__format__ (function), 242
Fraction, 31
fromhex (function), 126, 128
fromkeys (function), 138
frozenset (class), 130
frozenset (type), 9
fullmatch (function), 170, 172, 188
function, 14, 63
    __add__, 236
    __and__, 236
    __bool__, 243
    __bytes__, 242
    __call__, 244
    __complex__, 243
    __del__, 242
    __eq__, 239
    __float__, 243
    __floordiv__, 236
    __format__, 242
    __ge__, 239
    __getitem__, 24
    __gt__, 239
    __hash__, 242
    __iadd__, 236
    __iand__, 236
    __ifloordiv__, 236
    __ilshift__, 236
    __imod__, 236
    __imul__, 236

__init__, 215
__int__, 243
__ior__, 236
__ipow__, 236
__irshift__, 236
__isub__, 236
__iter__, 24
__itruediv__, 236
__ixor__, 236
__le__, 239
__lshift__, 236
__lt__, 239
__mod__, 236
__mul__, 236
__ne__, 239
__neg__, 236
__new__, 217
__next__, 24
__not__, 236
__or__, 236
__pos__, 236
__pow__, 236
__radd__, 239
__rand__, 239
__repr__, 231
__rfloordiv__, 239
__rlshift__, 239
__rmod__, 239
__rmul__, 239
__ror__, 239
__rpow__, 239
__rrshift__, 239
__rshift__, 236
__rsub__, 239
__rtruediv__, 239
__rxor__, 239
__str__, 231
__sub__, 236
__truediv__, 236
__xor__, 236
_asdict, 144
_fields, 144
_replace, 144
accumulate, 78, 82
add, 131
add_note, 205
all, 96
append, 123, 150
appendleft, 150
bin, 34, 54
bit_length, 11
callable, 49
capitalize, 160
center, 160
clear, 124, 136, 140, 152
close, 109
cmp_to_key, 85
compile, 170, 186
copy, 48, 124, 140, 152
count, 115, 162
decorator, 85
deepcopy, 13, 48
difference, 134
difference_update, 136
dir, 18
discard, 131
dropwhile, 113
elements, 148
end, 172
endswith, 162
enumerate, 25
escape, 191
extend, 123, 151
extendleft, 151
filter, 78, 79
find, 45, 162
findall, 189
finditer, 175, 178, 190
fromhex, 126, 128
fromkeys, 138
fullmatch, 170, 172, 188
generator, 93
get, 140
getattr, 143
getsizeof, 103
group, 172, 193
groupby, 116
groupdict, 194
groups, 172, 193
hex, 34, 126
higher order, 64
id, 10
index, 44, 152, 162
indices, 47
input, 3
insert, 122, 151
intersection, 134
intersection_update, 136
isalnum, 160
isalpha, 160
isascii, 160
isdecimal, 160

isdigit, 160
isdisjoint, 132
isidentifier, 160
iskeyword, 160
islice, 114
islower, 160
isnumeric, 160
isprintable, 160
isspace, 160
issubset, 132
issuperset, 133
istitle, 160
isupper, 160
items, 142
iter, 24, 93, 139
join, 164
keys, 142
len, 120, 138
ljust, 160
lower, 161
lstrip, 161
map, 78, 80
match, 187
matcher methods, 192
most_common, 148
namedtuple, 143
next, 24
oct, 34
partition, 163
pop, 124, 131, 140, 151
popitem, 141
popleft, 151
pow, 49
print, 4
purge, 192
range, 26
read, 127
reduce, 78, 80
remove, 123, 131, 151
removeprefix, 161
removesuffix, 161
replace, 162
reverse, 124, 152
reversed, 141
rfind, 162
rindex, 162
rjust, 160
rotate, 152
rpartition, 163
rsplit, 163
rstrip, 161
search, 186
setdefault, 140
sort, 78, 83
sorted, 74, 78, 83
span, 172
split, 163, 168, 188
splitlines, 163
start, 172
startswith, 162
strip, 161
sub, 170, 177, 190
subn, 191
subtract, 149
swapcase, 161
symmetric_difference, 135
symmetric_difference_update, 136
takewhile, 113
title, 160
to_bytes, 55
total, 149
type, 10
union, 133
update, 135, 141, 148
upper, 161
values, 142
variadic, 71
vars, 221
wrap, 87
write, 127
zfill, 161

funkcja
- count, 152
- end, 194
- next, 93
- span, 194
- start, 194

__ge__ (function), 239

generator
- pipe, 109

generator comprehension, 102
generator expression, 102
generator function, 93, 99
GeneratorExit (class), 207
get (function), 140
getattr (function), 143
__getitem__ (function), 24
getsizeof (function), 103
greedy quantifier, 168
group, 171

group (function), 172, 193
groupby (function), 116
groupdict (function), 194
groups (function), 172, 193
__gt__ (function), 239

hailstone sequence, 61
hash, 129
__hash__ (function), 129, 242
hash (function), 129
hashable, 129
Haskell, 2
hex (function), 34, 126
higher order function, 64

__iadd__ (function), 236
__iand__ (function), 236
id (function), 10
identity comparison operator, 58
IDLE, 5
if (statement), 22
if-else (expression), 22
__ifloordiv__ (function), 236
IGNORECASE, 184
__ilshift__ (function), 236
immutable, 12
__imod__ (function), 236
import, 16
__imul__ (function), 236
in (operator), 45, 58
indentation, 4
index (function), 44, 152, 162
indexing, 43
indices (function), 47
infinity, 37
__init__ (function), 215
inline flag, 181
input (function), 3
insert (function), 122, 151
instance attribute, 220
instance method, 222
instruction
    break, 23
    continue, 23
int, 33
__int__ (function), 243
int (type), 7
intersection (function), 134
intersection_update (function), 136
__ior__ (function), 236
__ipow__ (function), 236
__irshift__ (function), 236
is (operator), 13, 58
is not (operator), 58
isalnum (function), 160
isalpha (function), 160
isascii (function), 160
isdecimal (function), 160
isdigit (function), 160
isdisjoint (function), 132
isidentifier (function), 160
iskeyword (function), 160
islice (function), 114
islower (function), 160
isnumeric (function), 160
isprintable (function), 160
isspace (function), 160
issubset (function), 132
issuperset (function), 133
istitle (function), 160
__isub__ (function), 236
isupper (function), 160
items (function), 142
__iter__ (function), 24
iter (function), 24, 93, 139
iterable, 93
Iterable (protocol), 118
iterable object, 24
iterator, 24, 93
itertools (module), 113
__itruediv__ (function), 236
__ixor__ (function), 236

Java, 1
join (function), 164

KeyboardInterrupt (class), 207
keys (function), 142
keyword arguments, 69
Kleene, Stephen C, 165

lambda, 74
lastgroup, 194
lastindex, 194
lazy evaluation, 99
__le__ (function), 239
LEGB rule, 66
len (function), 120, 138
Lisp, 2, 74
list, 12
list (type), 7
list comprehension, 94
ljust (function), 160

lookahead assertion, 178
lookbehind assertion, 178
loop
    for, 24
    while, 23
lower (function), 161
__lshift__ (function), 236
lstrip (function), 161
__lt__ (function), 239

magic method, 49
magic methods, 234, 242
managed attribute, 230
map (function), 78, 80
mapping, 137
match (function), 187
match object, 170, 171, 186
match statement, 27
matcher, 186
matcher methods, 192
member, 211
membership test operator, 58
memoization, 88
method, 215
    __call__, 49
    class, 211, 223
    dunder, 49, 234
    instance, 222
    magic, 49
    non-static, 211
    static, 211, 225
ML, 2
__mod__ (function), 236
modifying operations, 121
Modula-3, 1
module
    itertools, 113
modulo (operator), 52
most_common (function), 148
__mul__ (function), 236
MULTILINE, 183
mutable, 12
MutableSequence (protocol), 118

namedtuple (function), 143
NaN, 37
__ne__ (function), 239
__neg__ (function), 236
negation (logical), 36
nested decorator, 89
__new__ (function), 217
__next__ (function), 24
next (function), 24
next (funkcja), 93
non-modifying operations, 118
None, 31
NoneType type, 31
nonlocal, 65
__not__ (function), 236
not (operator), 36, 59
not a number, 37
not in (operator), 45, 58
NotImplementedType (type), 32

oct (function), 34
operator, 40
    **, 12
    and, 36, 59
    arithmetic, 50
    arithmetic comparison, 57
    bitwise, 53
    bitwise and, 56
    bitwise not, 56
    bitwise or, 56
    bitwise xor, 56
    boolean, 59
    call, 48
    comparison, 57
    division, 51
    exponentiation, 12, 49
    identity comparison, 58
    in, 45, 58
    is, 13, 58
    is not, 58
    membership test, 58
    modulo, 52
    not, 36, 59
    not in, 45, 58
    or, 36, 59
    overloading, 234
    precedence, 42
    remainder, 52
    shift, 54
    slicing, 43
    subscription, 43
    unary, 50
    unpacking, 155
option
    ASCII, 184
    DOTALL, 184
    IGNORECASE, 184
    MULTILINE, 183

VERBOSE, 185
option flags, 181
__or__ (function), 236
or (operator), 36, 59
overloading, 234

parenthesized expression, 42
partition (function), 163
pass statement, 21
Pattern (class), 170, 186
PEP, 6
8, 6
20, 5
636, 30
Peters, Tim, 83
pipes, 109
pop (function), 124, 131, 140, 151
popitem (function), 141
popleft (function), 151
__pos__ (function), 236
positional parameter, 69
possessive quantifier, 168
__pow__ (function), 236
pow (function), 49
precedence, 42
print (function), 4
property, 227
protocol
Collection, 118
Container, 118
Iterable, 118
MutableSequence, 118
Reversible, 118
Sized, 118
purge (function), 192
PyPI, 2
Python shell, 2

quantifier, 168
greedy, 168
possessive, 168
reluctant, 168

__radd__ (function), 239
raise, 198, 201
__rand__ (function), 239
range (function), 26
range (type), 8
raw string, 39, 159, 168
re (module), 165
read (function), 127
recursive lambdas, 78
reduce (function), 78, 80
regex, 165
object, 186
pattern, 186
regex (module), 165
regular expression, 165
reluctant quantifier, 168
remainder (operator), 52
remove (function), 123, 131, 151
removeprefix (function), 161
removesuffix (function), 161
REPL, 2
replace (function), 162
__repr__ (function), 231
reverse (function), 152
reversed (function), 141
Reversible (protocol), 118
rfind (function), 162
__rrfloordiv__ (function), 239
rindex (function), 162
rjust (function), 160
__rlshift__ (function), 239
__rmod__ (function), 239
__rmul__ (function), 239
__ror__ (function), 239
van Rossum, Guido, 1, 74
rotate (function), 152
rpartition (function), 163
__rpow__ (function), 239
__rrshift__ (function), 239
__rshift__ (function), 236
rsplit (function), 163
rstrip (function), 161
__rsub__ (function), 239
__rtruediv__ (function), 239
__rxor__ (function), 239

scope, 65
search (function), 186
self, 211
sequential collection, 118
set (class), 130
set (type), 9
set comprehension, 97
setdefault (function), 140
shebang, 11
shift (operator, 54
Sized (protocol), 118
slice, 45
slicing, 43
sort (function), 78, 83

sorted (function), 74, 78, 83
span (function), 172
span (funkcja), 194
split (function), 163, 168, 188
splitlines (function), 163
start (function), 172
start (funkcja), 194
startswith (function), 162
statement
  for loop, 24
  if, 22
  match, 27
  pass, 21
  while loop, 23
static attribute, 220
static field, 220
static method, 211, 225
StopIteration, 100
StopIteration (exception), 24, 93
__str__ (function), 231
str (type), 38
string
  raw, 39, 168
  triple quote, 39
strip (function), 161
strong typing, 1
__sub__ (function), 236
sub (function), 170, 177, 190
subn (function), 191
subscription (operator), 43
subtract (function), 149
swapcase (function), 161
symmetric_difference (function), 135
symmetric_difference_update (function), 136
SystemExit (class), 207

takewhile (function), 113
Thompson, Ken, 165
title (function), 160
to_bytes (function), 55
total (function), 149
total_ordering (decorator), 241
triple quote string, 39
True, 35
__truediv__ (function), 236
truthy, 35
try clause, 198
tuple (type), 8
type
  bool, 35
  complex, 7, 37
  dict, 9
  ellipsis, 32
  float, 7, 36
  frozenset, 9
  int, 7, 33
  list, 7
  NoneType, 31
  NotImplementedType, 32
  range, 8
  set, 9
  str, 38
  tuple, 8
type (class), 211
type (function), 10
typing
  dynamic, 1
  strong, 1

unary operator, 50
union (function), 133
unpacking iterables, 153
update (function), 135, 141, 148
upper (function), 161

values (function), 142
variadic function, 71
vars (function), 221
VERBOSE, 185

Wall, Larry, 165
walrus, 41
while loop, 23
wrap (function), 87
write (function, 127

__xor__ (function), 236

zfill (function), 161