Python
Tutorial

by
Bernd Klein

Python Course
Python Tutorial by
Bernd Klein

EASY AS ABC

What do the alphabet and the programming language Python have in common? Right, both start with ABC. If we are talking about ABC in the Python context, it's clear that the programming language ABC is meant. ABC is a general-purpose programming language and programming environment, which was developed in the Netherlands, Amsterdam, at the CWI (Centrum Wiskunde & Informatica). The greatest achievement of ABC was to influence the design of Python.

Python was conceptualized in the late 1980s. Guido van Rossum worked that time in a project at the CWI, called Amoeba, a distributed operating system. In an interview with Bill Venners1, Guido van Rossum said: "In the early 1980s, I worked as an implementer on a team building a language called ABC at Centrum voor Wiskunde en Informatica (CWI). I don't know how well people know ABC's influence on Python. I try to mention ABC's influence because I'm indebted to everything I learned during that project and to the people who worked on it."

Later on in the same Interview, Guido van Rossum continued: "I remembered all my experience and some of my frustration with ABC. I decided to try to design a simple scripting language that possessed some of ABC's better properties, but without its problems. So I started typing. I created a simple virtual machine, a simple parser, and a simple runtime. I made my own version of the various ABC parts that I liked. I created a basic syntax, used indentation for statement grouping instead of curly braces or begin-end blocks, and developed a small number of powerful data types: a hash table (or dictionary, as we call it), a list, strings, and numbers."

COMEDY, SNAKE OR PROGRAMMING LANGUAGE

So, what about the name "Python"? Most people think about snakes, and even the logo depicts two snakes, but the origin of the name has its root in British humour. Guido van Rossum, the creator of Python, wrote in 1996 about the origin of the name of his programming language1: "Over six years ago, in December 1989, I was looking for a 'hobby' programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

THE ZEN OF PYTHON

```
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 d
o 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!
```

DEVELOPMENT STEPS OF PYTHON

Guido Van Rossum published the first version of Python code (version 0.9.0) at alt.sources in February 1991. This release included already exception handling, functions, and the core data types of list, dict, str and others. It was also object oriented and had a module system.

Python version 1.0 was released in January 1994. The major new features included in this release were the functional programming tools lambda, map, filter and reduce, which Guido Van Rossum never liked.

Six and a half years later in October 2000, Python 2.0 was introduced. This release included list comprehensions, a full garbage collector and it was supporting unicode.

Python flourished for another 8 years in the versions 2.x before the next major release as Python 3.0 (also known as "Python 3000" and "Py3K") was released. Python 3 is not backwards compatible with Python 2.x. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules, thus fulfilling or coming close to fulfilling the 13th law of the Zen of Python: "There should be one -- and preferably only one -- obvious way to do it."

SOME CHANGES IN PYTHON 3.0:

```
Print is now a function
Views and iterators instead of lists
The rules for ordering comparisons have been simplified. E.g. a het
erogeneous list cannot be sorted, because all the elements of a lis
t must be comparable to each other.
There is only one integer type left, i.e. int. long is int as well.
The division of two integers returns a float instead of an intege
r. "//" can be used to have the "old" behaviour.
Text Vs. Data Instead Of Unicode Vs. 8-bit
```

THE INTERPRETER, AN INTERACTIVE SHELL

THE TERMS 'INTERACTIVE' AND 'SHELL'

The term "interactive" traces back to the Latin expression "inter agere". The verb "agere" means amongst other things "to do something" and "to act", while "inter" denotes the spatial and temporal position to things and events, i.e. "between" or "among" objects, persons, and events. So "inter agere" means "to act between" or "to act among" these.

With this in mind, we can say that the interactive shell is between the user and the operating system (e.g. Linux, Unix, Windows or others). Instead of an operating system an interpreter can be used for a programming language like Python as well. The Python interpreter can be used from an interactive shell.

The interactive shell is also interactive in the way that it stands between the commands or actions and their execution. In other words, the shell waits for commands from the user, which it executes and returns the result of the execution. Afterwards, the shell waits for the next input.

A shell in biology is a calcium carbonate "wall" which protects snails or mussels from its environment or its enemies. Similarly, a shell in operating systems lies between the kernel of the operating system and the user. It's a "protection" in both directions. The user doesn't have to use the complicated basic functions of the OS but is capable of using the interactive shell which is relatively simple and easy to understand. The kernel is protected from unintentional and incorrect usages of system functions.

Python offers a comfortable command line interface with the Python Shell, which is also known as the "Python Interactive Shell". It looks like the term "Interactive Shell" is a tautology, because "shell" is interactive on its own, at least the kind of shells we have described in the previous paragraphs.

USING THE PYTHON INTERACTIVE SHELL

With the Python interactive interpreter it is easy to check Python commands. The Python interpreter can be invoked by typing the command "python" without any parameter followed by the "return" key at the shell prompt:

```
python
```

Python comes back with the following information:

```
$ python
Python 2.7.11 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
```

```
    Type "help", "copyright", "credits" or "license" for more informati
    on.
```

A closer look at the output above reveals that we have the wrong Python version. We wanted to use Python 3.x, but what we got is the installed standard of the operating system, i.e. version 2.7.11+.

The easiest way to check if a Python 3.x version is installed: Open a Terminal. Type in python but no return. Instead, type the "Tab" key. You will see possible extensions and other installed versions, if there are some:

```
bernd@venus:~$ $ python
python            python3           python3.7-config   python-con
fig
python2           python3.6         python3.7m         pythontex
python2.7         python3.6-config  python3.7m-config  pythontex3
python2.7-config  python3.6m        python3-config     python-whi
teboard
python2-config    python3.6m-config python3m
python2-pbr       python3.7         python3m-config
bernd@venus:~$ python
```

If no other Python version shows up, python3.x has to be installed. Afterwards, we can start the newly installed version by typing python3:

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more informati
on.
```

Once the Python interpreter is started, you can issue any command at the command prompt ">>>". Let's see, what happens, if we type in the word "hello":

```
hello
```

```
-------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-f572d396fae9> in <module>
----> 1 hello

NameError: name 'hello' is not defined
```

Of course, "hello" is not a proper Python command, so the interactive shell returns ("raises") an error.

The first real command we will use is the print command. We will create the mandatory "Hello World" statement:

```
print("Hello World")
```

```
Hello World
```

It couldn't have been easier, could it? Oh yes, it can be written in a even simpler way. In the Interactive Python Interpreter - but not in a program - the print is not necessary. We can just type in a string or a number and it will be "echoed"

```
"Hello World"
```
Output: `'Hello World'`

```
3
```
Output: 3

HOW TO QUIT THE PYTHON SHELL

So, we have just started, and we are already talking about quitting the shell. We do this, because we know, how annoying it can be, if you don't know how to properly quit a program.

It's easy to end the interactive session: You can either use exit() or Ctrl-D (i.e. EOF) to exit. The brackets behind the exit function are crucial. (Warning: Exit without brackets works in Python2.x but doesn't work anymore in Python3.x)

THE SHELL AS A SIMPLE CALCULATOR

In the following example we use the interpreter as a simple calculator by typing an arithmetic expression:

```
4.567 * 8.323 * 17
```
Output: 646.189397

Python follows the usual order of operations in expressions. The standard order of operations is expressed in the following enumeration:

- exponents and roots
- multiplication and division
- addition and subtraction

In other words, we don't need parentheses in the expression "3 + (2 * 4)":

```
3 + 2 * 4
```
Output: 11

The most recent output value is automatically stored by the interpreter in a special variable with the name "_". So we can print the output from the recent example again by typing an underscore after the prompt:

```
_
```
Output: 11

The underscore can be used in other expressions like any other variable:

```
_ * 3
```
Output: 33

The underscore variable is only available in the Python shell. It's NOT available in Python scripts or programs.

USING VARIABLES

It's simple to use variables in the Python shell. If you are an absolute beginner and if you don't know anything about variables, please refer to our chapter about variables and data types. Values can be saved in variables. Variable names don't require any special tagging, like they do in Perl, where you have to use dollar signs, percentage signs and at signs to tag variables:

```
maximal = 124
width = 94
print(maximal - width)
```

```
30
```

MULTILINE STATEMENTS

We haven't introduced multiline statements so far. So beginners can skip the rest of this chapter and can continue with the following chapters.

We will show how the interactive prompt deals with multiline statements like for loops.

```
l = ["A", 42, 78, "Just a String"]
for character in l:
    print(character)
```

```
A
42
78
Just a String
```

After having input "for character in l: " the interpretor expects the input of the next line to be indented. In other words: the interpretor expects an indented block, which is the body of the loop. This indented block will be iterated. The interpretor shows this "expectation" by displaying three dots "..." instead of the standard Python interactive prompt ">>>". Another special feature of the interactive shell: when we are done with the indented lines, i.e. the block, we have to enter an empty line to indicate that the block is finished. Attention: the additional empty line is only necessary in the interactive shell! In a Python program, it is enough to return to the indentation level of the "for" line, the one with the colon ":" at the end.

STRINGS

Strings are created by putting a sequence of characters in quotes. Strings can be surrounded by single quotes, double quotes or triple quotes, which are made up of three single or three double quotes. Strings are immutable. In other words, once defined, they cannot be changed. We will cover this topic in detail in another chapter.

```
"Hello" + " " + "World"
```
Output: `'Hello World'`

A string in triple quotes can span several lines without using the escape character:

```
city = """
... Toronto is the largest city in Canada
... and the provincial capital of Ontario.
... It is located in Southern Ontario on the
... northwestern shore of Lake Ontario.
... """
print(city)
```

```
Toronto is the largest city in Canada
and the provincial capital of Ontario.
It is located in Southern Ontario on the
northwestern shore of Lake Ontario.
```

Multiplication on strings is defined, which is essentially a multiple concatenation:

```
".-." * 4
```
Output: `'.-..-..-..-.'`

EXECUTE A PYTHON SCRIPT

So far we have played around with Python commands in the Python shell. We want to write now our first serious Python program. You will hardly find any beginner's textbook on programming, which doesn't start with the "almost mandatory" "Hello World" program, i.e. a program which prints the string "Hello World". We start a Python interactive shell in a terminal with the command "python". It might be necessary to use "python3" to get a Python3 shell:

```
$ python3
(base) bernd@moon:~$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more informati
on.
>>> print("Hello World!")
Hello World!
>>>
```

But, as we said at the beginning, we want to write a "serious" program now, i.e. a program which resides in a file. This way, we can use a program over and over again without having to type it in again. Some may like to call such a little program "script". We will use a slight variation of the "Hello World" theme. We have to include our print function into a file. To save and edit our program in a file we need an editor. There are lots of editors, but you should choose one, which supports syntax highlighting and indentation. For Linux you can use vi, vim, emacs, geany, gedit and umpteen others. The Emacs works for Windows as well, but Notepad++ may be the better choice in many cases. Of course, you can also use an IDE (Integrated Development Environment) like PyCharm or Spyder for this purpose.

So, after you have found the editor or the IDE of your choice, you are ready to develop your mini program, i.e.

```
print("My first simple Python script!")
```

```
My first simple Python script!
```
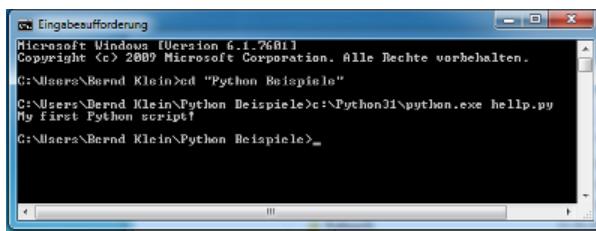
and save it as my_first_simple_program.py. The suffix .py is not really necessary for Linux but it's good to use it since the extension is essential, if you want to write modules.

Let's assume our script is in a subdirectory under the home directory of the user monty:

```
monty@python:~$ cd python
monty@python:~/python$ python my_first_simple_program.py
My first simple Python script!
monty@python:~/python$
```

It can be started in a Command Prompt in Windows as well (start -> All Programs -> Accessories -> Command Prompt). You may notice that we called our little program "hello.py" and not my_first_simple_program.py:

STARTING A PYTHON SCRIPT UNDER WINDOWS

PYTHON INTERNALS

Most probably you have read somewhere that the Python language is an interpreted programming or a script language. The truth is: Python is both an interpreted and a compiled language. But calling Python a compiled language would be misleading. (At the end of this chapter, you will find the definitions for Compilers and Interpreters, in case you are not already familiar with the concepts!) People would assume that the compiler translates the Python code into machine language. Python code is translated into intermediate code, which has to be executed by a virtual machine, known as the PVM, the Python Virtual Machine. This is a similar approach to the one taken by Java. There is even a way of translating Python programs into Java byte code for the Java Virtual Machine (JVM). This can be achieved with Jython.

The question is, do I have to compile my Python scripts to make them faster or how can I compile them? The answer is easy: normally, you don't need to do anything and you shouldn't bother, because "Python" is already doing the thinking for you, i.e. it takes the necessary steps automatically.

For whatever reason you want to compile a python program manually? No problem. It can be done with the module py_compile, either using the interpreter shell

```python
import py_compile
py_compile.compile('my_first_simple_program.py')
```
Output: '__pycache__/my_first_simple_program.cpython-37.pyc'

or using the following command at the shell prompt

```
python -m py_compile my_first_simple_program.py
```
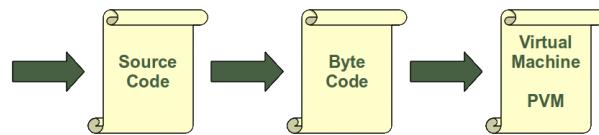
Either way, you may notice two things: first, there will be a new subdirectory "__pycache__", if it doesn't already exist. You will find a file "my_first_simple_script.cpython-34.pyc" in this subdirectory. This is the compiled version of our file in byte code.

You can also automatically compile all Python files using the compileall module. You can do it from the shell prompt by running compileall.py and providing the path of the directory containing the Python files to compile:

```
monty@python:~/python$ python -m compileall .
Listing . ...
```

But as we have said, you don't have to and shouldn't bother about compiling Python code. The compilation is hidden from the user for a good reason. Some newbies wonder sometimes where these ominous files with the .pyc suffix might come from. If Python has write-access for the directory where the Python program resides, it will store the compiled byte code in a file that ends with a .pyc suffix. If Python has no write access, the

program will work anyway. The byte code will be produced but discarded when the program exits. Whenever a Python program is called, Python will check, if a compiled version with the .pyc suffix exists. This file has to be newer than the file with the .py suffix. If such a file exists, Python will load the byte code, which will speed up the start up time of the script. If there is no byte code version, Python will create the byte code before it starts the execution of the program. Execution of a Python program means execution of the byte code on the Python.

COMPILATION OF A PYTHON SCRIPT

Every time a Python script is executed, a byte code is created. If a Python script is imported as a module, the byte code will be stored in the corresponding .pyc file. So, the following will not create a byte code file:

```
monty@python:~/python$ python my_first_simple_program.py
My first simple Python script!
monty@python:~/python$
```

The import in the following Python2 session will create a byte code file with the name "my_first_simple_program.pyc":

```
monty@python:~/tmp$ ls
my_first_simple_program.py
monty@python:~/tmp$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
>>> import my_first_simple_script
My first simple Python script!
>>> exit()
monty@python:~/tmp$ ls
my_first_simple_program.py  my_first_simple_program.pyc
monty@python:~/tmp$
```

RUNNABLE SCRIPTS UNDER LINUX

This chapter can be skipped by Windows users. Don't worry! It is not essential!

So far, we have started our Python scripts with

```
python3 my_file.py
```

on the bash command line. A Python script can also be started like any other script under Linux, e.g. Bash scripts. Two steps are necessary for this purpose: the shebang line #!/usr/bin/env python3 has to be added as the first line of your Python code file. Alternatively, this line can be #!/usr/bin/python3, if this is the location of your Python interpreter. Instead using env as in the first shebang line, the interpreter is searched for and located at the time the script is run. This makes the script more portable. Yet, it also suffers from the same problem: The path to env may also be different on a per-machine basis. The file has to be made executable: The command "chmod +x scriptname" has to be executed on a Linux shell, e.g. bash. "chmod 755 scriptname" can also be used to make your file executable. In our example:

```
$ chmod +x my_first_simple_program.py
```

We illustrate this in a bash session:

```
bernd@marvin:~$ more my_first_simple_script.py
#!/usr/bin/env python3
print("My first simple Python script!")
bernd@marvin:~$ ls -ltr my_first_simple_script.py
-rw-r--r-- 1 bernd bernd 63 Nov  4 21:17 my_first_simple_script.py
bernd@marvin:~$ chmod +x my_first_simple_script.py
bernd@marvin:~$ ls -ltr my_first_simple_script.py
-rwxr-xr-x 1 bernd bernd 63 Nov  4 21:17 my_first_simple_script.py
bernd@marvin:~$ ./my_first_simple_script.py
My first simple Python script!
```

DIFFERENCES BETWEEN COMPILERS AND INTERPRETERS

COMPILER

Definition: a compiler is a computer program that transforms (translates) source code of a programming language (the source language) into another computer language (the target language). In most cases compilers are used to transform source code into executable program, i.e. they translate code from high-level programming languages into low (or lower) level languages, mostly assembly or machine code.

INTERPRETER

Definition: an interpreter is a computer program that executes instructions written in a programming language. It can either execute the source code directly or translate the source code in a first step into a more efficient representation and execute this code.

STRUCTURING WITH INDENTATION

BLOCKS

A block is a group of statements in a program or script. Usually, it consists of at least one statement and declarations for the block, depending on the programming or scripting language. A language which allows grouping with blocks, is called a block structured language. Generally, blocks can contain blocks as well, so we get a nested block structure. A block in a script or program functions as a means to group statements to be treated as if they were one statement. In many cases, it also serves as a way to limit the lexical scope of variables and functions.

Initially, in simple languages like Basic and Fortran, there was no way of explicitly using block structures. Programmers had to rely on "go to" structures, nowadays frowned upon, because "Go to programs" turn easily into spaghetti code, i.e. tangled and inscrutable control structures.

Block structures were first formalized in ALGOL as a compound statement.

Programming languages, such as ALGOL, Pascal, and others, usually use certain methods to group statements into blocks:

- begin ... end

A code snippet in Pascal to show this usage of blocks:

```
with ptoNode^ do
    begin
      x := 42;
      y := 'X';
    end;
```

- do ... done

- if ... fi e.g. Bourne and Bash shell

- Braces (also called curly brackets): { ... } By far the most common approach, used by C, C++, Perl, Java, and many other programming languages, is the use of braces. The following example shows a conditional statement in C:

```
if (x==42) {
    printf("The Answer to the Ultimate Question of Life, the Univer
```

```
    se, and Everything\n");
    } else {
        printf("Just a number!\n");
    }
```

The indentations in this code fragment are not necessary. So the code could be written - offending common decency - as

```
 if (x==42) {printf("The  Answer to the Ultimate Question of Life,
 the Universe, and Everything\n");} else {printf("Just a numbe
 r!\n");}
```
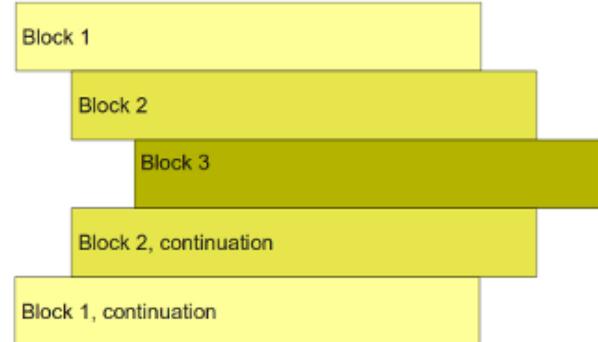
Please, keep this in mind to understand the advantages of Python!

INDENTING CODE

Python uses a different principle. Python programs get structured through indentation, i.e. code blocks are defined by their indentation. Okay that's what we expect from any program code, isn't it? Yes, but in the case of Python it's a language requirement, not a matter of style. This principle makes it easier to read and understand other people's Python code.

So, how does it work? All statements with the same distance to the right belong to the same block of code, i.e. the statements within a block line up vertically. The block ends at a line less indented or the end of the file. If a block has to be more deeply nested, it is simply indented further to the right.

Beginners are not supposed to understand the following example, because we haven't introduced most of the used structures, like conditional statements and loops. Please see the following chapters about loops and conditional statements for explanations. The program implements an algorithm to calculate Pythagorean triples. You will find an explanation of the Pythagorean numbers in our chapter on for loops.

```python
from math import sqrt
n = input("Maximum Number? ")
n = int(n)+1
for a in range(1,n):
    for b in range(a,n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
```

```
            print(a, b, c)
```

```
3  4  5
5  12  13
6  8  10
8  15  17
9  12  15
12  16  20
15  20  25
```

There is another aspect of structuring in Python, which we haven't mentioned so far, which you can see in the example. Loops and Conditional statements end with a colon ":" - the same is true for functions and other structures introducing blocks. All in all, Python structures by colons and indentation.

DATA TYPES AND VARIABLES

INTRODUCTION

Have you programmed low-level languages like C, C++ or other similar programming languages? If so, you might think you know already enough about data types and variables. You know a lot, that's right, but not enough for Python. So it's worth to go on reading this chapter on data types and variables in Python. There are dodgy differences in the way Python and C deal with variables. There are integers, floating point numbers, strings, and many more, but things are not the same as in C or C++.

If you want to use lists or associative arrays in C e.g., you will have to construe the data type list or associative arrays from scratch, i.e., design memory structure and the allocation management. You will have to implement the necessary search and access methods as well. Python provides power data types like lists and associative arrays called "dict", as a genuine part of the language.
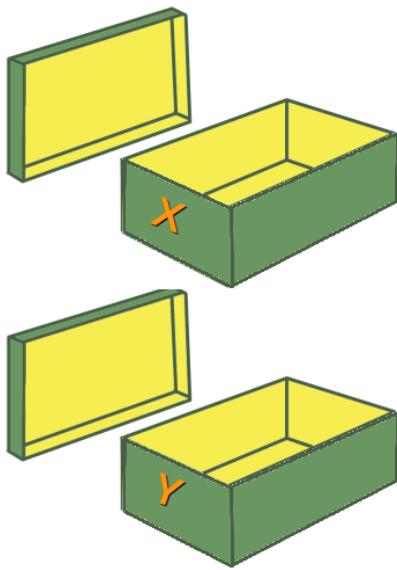
So, this chapter is worth reading both for beginners and for advanced programmers in other programming languages.

VARIABLES

GENERAL CONCEPT OF VARIABLES

This subchapter is especially intended for C, C++ and Java programmers, because the way these programming languages treat basic data types is different from the way Python does. Those who start learning Python as their first programming language may skip this and read the next subchapter.

So, what's a variable? As the name implies, a variable is something which can change. A variable is a way of referring to a memory location used by a computer program. In many programming languages a variable is a symbolic name for this physical location. This memory location contains values, like numbers, text or more complicated types. We can use this variable to tell the computer to save some data in this location or to retrieve some data from this location.

A variable can be seen as a container (or some say a pigeonhole) to store certain values. While the program is running, variables are accessed and sometimes changed, i.e., a new value will be assigned to a variable.
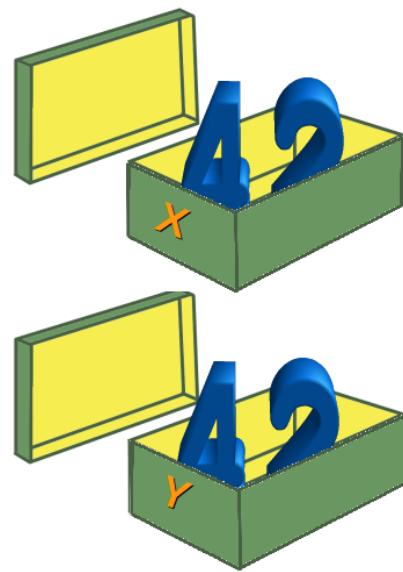
What we have said so far about variables best fits the way variables are implemented in C, C++ or Java. Variable names have to be declared in these languages before they can be used.

```
int x;

int y;
```

Such declarations make sure that the program reserves memory for two variables with the names x and y. The variable names stand for the memory location. It's like the two empty shoeboxes, which you can see in the picture above. These shoeboxes are labeled with x and y. Like the two shoeboxes, the memory is empty as well.

Putting values into the variables can be realized with assignments. The way you assign values to variables is nearly the same in all programming languages. In most cases, the equal "=" sign is used. The value on the right side will be saved in the variable name on the left side.

We will assign the value 42 to both variables and we can see that two numbers are physically saved in the memory, which correspond to the two shoeboxes in the following picture.
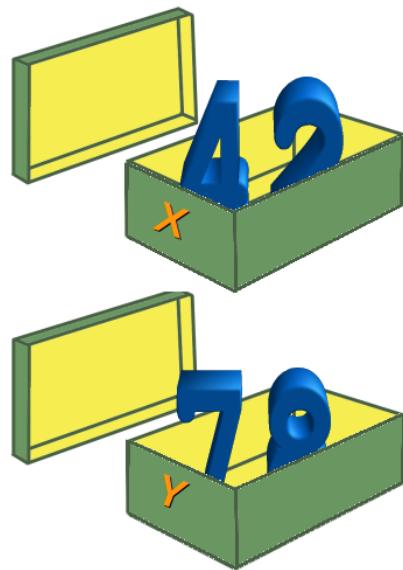
```
x = 42;

y = 42;
```

We think that it is easy to understand what happens. If we assign a new value to one of the variables, let's say the value 78 to y:

```
y = 78;
```

We have exchanged the content of the memory location of y.

We have seen now that in languages like C, C++ or Java every variable has and must have a unique data type. E.g., if a variable is of type integer, solely integers can be saved in the variable for the duration of the program. In those programming languages every variable has to be declared before it can be used. Declaring a variable means binding it to a data type.

There is no declaration of variables required in Python, which makes it quite easy. It's not even possible to declare the variables. If there is need for a variable, you should think of a name and start using it as a variable.

Another remarkable aspect of Python: Not only the value of a variable may change during program execution, but the type as well. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the same variable. In the following line of code, we assign the value 42 to a variable:

```
i = 42
```

The equal "=" sign in the assignment shouldn't be seen as "is equal to". It should be "read" or interpreted as "is set to", meaning in our example "the variable i is set to 42". Now we will increase the value of this variable by 1:

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

 43
```
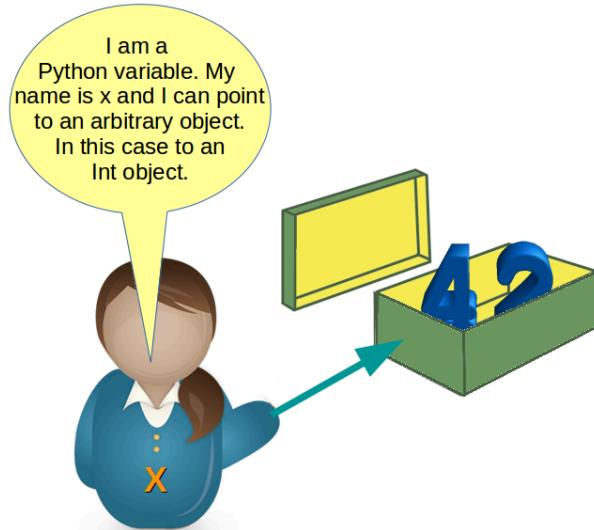
As we have said above, the type of a variable can change during the execution of a script. Or, to be precise, a new object, which can be of any type, will be assigned to it. We illustrate this in our following example:

```
    i = 42          # data type is implicitly set to integer
    i = 42 + 0.11     # data type is changed to float
    i = "forty"     # and now it will be a string
```

When Python executes an assignment like "i = 42", it evaluates the right side of the assignment and recognizes that it corresponds to the integer number 42. It creates an object of the integer class to save this data. If you want to have a better understanding of objects and classes, you can but you don't have to start with our chapter on Object-Oriented Programming.

In other words, Python automatically takes care of the physical representation for the different data types.

We want to take a closer look at variables now. Python variables are references to objects, but the actual data is contained in the objects:

As variables are pointing to objects and objects can be of arbitrary data types, variables cannot have types associated with them. This is a huge difference from C, C++ or Java, where a variable is associated with a fixed data type. This association can't be changed as long as the program is running.

Therefore it is possible to write code like the following in Python:
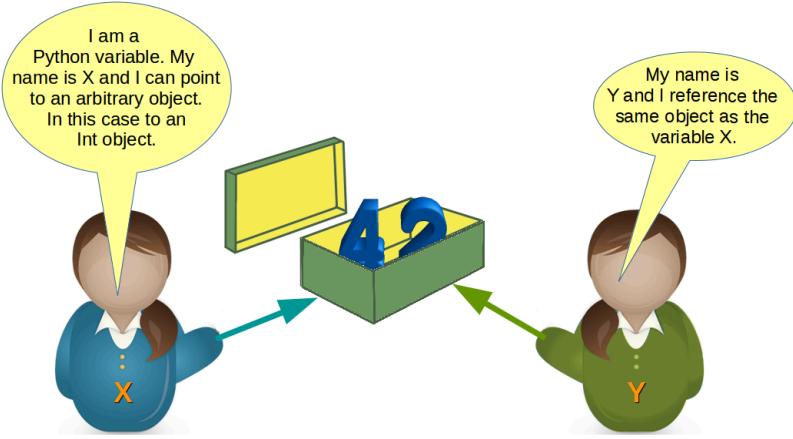
```
x = 42
print(x)

42

x = "Now x references a string"
print(x)

Now x references a string
```

We want to demonstrate something else now. Let's look at the following code:

```
x = 42
y = x
```

We created an integer object 42 and assigned it to the variable x. After this we assigned x to the variable y. This means that both variables reference the same object. The following picture illustrates this:
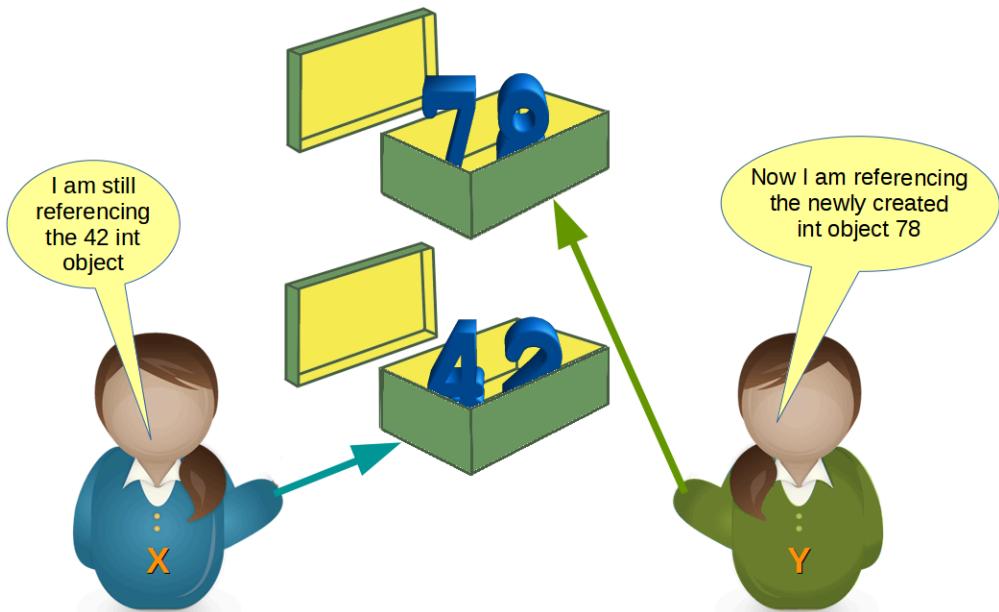
What will happen when we execute

```
y = 78
```
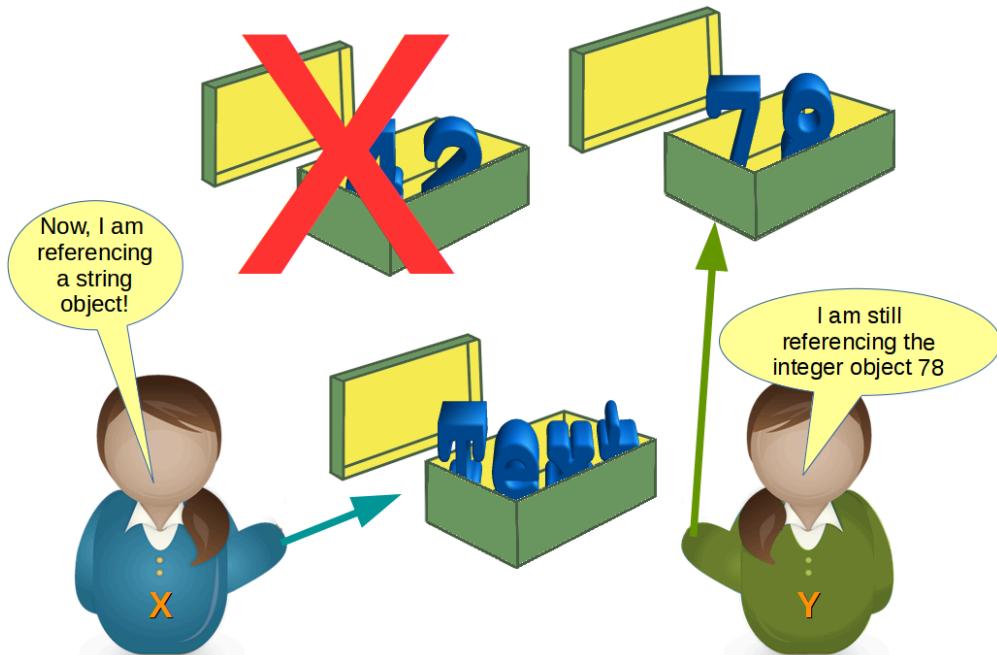
after the previous code?

Python will create a new integer object with the content 78 and then the variable y will reference this newly created object, as we can see in the following picture:

Most probably, we will see further changes to the variables in the flow of our program. There might be, for example, a string assignment to the variable x. The previously integer object "42" will be orphaned after this

assignment. It will be removed by Python, because no other variable is referencing it.

You may ask yourself, how can we see or prove that x and y really reference the same object after the assignment y = x of our previous example?

The identity function id() can be used for this purpose. Every instance (object or variable) has an identity, i.e., an integer which is unique within the script or program, i.e., other objects have different identities. So, let's have a look at our previous example and how the identities will change:

```
x = 42
id(x)
```
Output: 140709828470448

```
y = x
id(x), id(y)
```
Output: (140709828470448, 140709828470448)

```
y = 78
id(x), id(y)
```
Output: (140709828470448, 140709828471600)

VALID VARIABLE NAMES

The naming of variables follows the more general concept of an identifier. A Python identifier is a name used to identify a variable, function, class, module or other object.

A variable name and an identifier can consist of the uppercase letters "A" through "Z", the lowercase letters "a" through "z", the underscore _ and, except for the first character, the digits 0 through 9. Python 3.x is based on Unicode. That is, variable names and identifier names can additionally contain Unicode characters as well.

Identifiers are unlimited in length. Case is significant. The fact that identifier names are case-sensitive can cause problems to some Windows users, where file names are case-insensitive, for example.

Exceptions from the rules above are the special Python keywords, as they are described in the following paragraph.

The following variable definitions are all valid:

```
height = 10
maximum_height = 100

υψος = 10
μεγιστη_υψος = 100

MinimumHeight = 1
```

PYTHON KEYWORDS

No identifier can have the same name as one of the Python keywords, although they are obeying the above naming conventions:

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

There is no need to learn them by heart. You can get the list of Python keywords in the interactive shell by using help. You type help() in the interactive, but please don't forget the parenthesis:

```
help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely chec
k out
```

```
the tutorial on the Internet at http://docs.python.org/3.4/tutoria
l/.

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

To get a list of available modules, keywords, symbols, or topics, t
ype
"modules", "keywords", "symbols", or "topics".  Each module also co
mes
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 spa
m".

help>
```

What you see now is the help prompt, which allows you to query help on lots of things, especially on "keywords" as well:

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get mo
re help.

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

help>
```

NAMING CONVENTIONS

We saw in our chapter on "Valid Variable Names" that we sometimes need names which consist of more than one word. We used, for example, the name "maximum_height". The underscore functioned as a word separator, because blanks are not allowed in variable names. Some people prefer to write variable names in the

so-called CamelCase notation. We defined the variable MinimumHeight in this naming style.

There is a permanent "war" going on between the camel case followers and the underscore lovers. Personally, I definitely prefer "the_natural_way_of_naming_things" to "TheNaturalWayOfNamingThings". I think that the first one is more readable and looks more natural language like. In other words: CamelCase words are harder to read than their underscore counterparts, EspeciallyIfTheyAreVeryLong. This is my personal opinion shared by many other programmers but definitely not everybody. The Style Guide for Python Code recommends underscore notation for variable names as well as function names.

Certain names should be avoided for variable names: Never use the characters 'l' (lowercase letter "L"), 'O' ("O" like in "Ontario"), or 'I' (like in "Indiana") as single character variable names. They should be avoided, because these characters are indistinguishable from the numerals one and zero in some fonts. When tempted to use 'l', use 'L' instead, if you cannot think of a better name anyway. The Style Guide has to say the following about the naming of identifiers in standard modules: "All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names."

Companies, institutes, organizations or open source projects aiming at an international audience should adopt a similar notation convention!

CHANGING DATA TYPES AND STORAGE LOCATIONS

Programming means data processing. Data in a Python program is represented by objects. These objects can be

- built-in, i.e., objects provided by Python,
- objects from extension libraries,
- created in the application by the programmer.

So we have different "kinds" of objects for different data types. We will have a look at the different built-in data types in Python.

NUMBERS

Python's built-in core data types are in some cases also called object types. There are four built-in data types for numbers:

- Integer
  - Normal integers
  e.g., 4321
  - Octal literals (base 8)
  A number prefixed by 0o (zero and a lowercase "o" or uppercase "O") will be interpreted as an octal number
  example:

```
a = 0o10

print(a)
```

8

  •Hexadecimal literals (base 16)
  Hexadecimal literals have to be prefixed either by "0x" or "0
X".
  example:

```
hex_number = 0xA0F

print(hex_number)
```

2575

  •Binary literals (base 2)
  Binary literals can easily be written as well. They have to be
prefixed by a leading "0", followed by a "b" or "B":

```
x = 0b101010
x
```
Output: 42

The functions hex, bin, oct can be used to convert an integer number into the corresponding string representation of the integer number:

```
x = hex(19)
x
```
Output: '0x13'

```
type(x)
```
Output: str

```
x = bin(65)
x
```
Output: '0b1000001'

```
x = oct(65)
x
```

Output: `'0o101'`

```
oct(0b101101)
```

Output: `'0o55'`

Integers in Python3 can be of unlimited size

```
x = 787366098712738903245678234782358292837498729182728
print(x)
```

```
787366098712738903245678234782358292837498729182728
```

```
x * x * x
```

Output:
```
488123970070638215986770162105731315538827586091948617997871
229502288911239609019183086182863115232822393137082755897871
2300531714896856979787558109235
```

• Long integers

Python 2 has two integer types: int and long. There is no "long int" in Python3 anymore. There is only one "int" type, which contains both "int" and "long" from Python2. That's why the following code fails in Python 3:

```
    1L
  File "<stdin>", line 1
    1L
     ^
```

```
  File "<ipython-input-27-315518d611d8>", line 1
    1L
     ^
SyntaxError: invalid syntax
```

```
x = 43
long(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-49-bad7871d6042> in <module>
      1 x = 43
----> 2 long(x)

NameError: name 'long' is not defined
```

- Floating-point numbers

For example: 42.11, 3.1415e-10

- Complex numbers

Complex numbers are written as

```
<real part> + <imaginary part>
```

Examples:

```python
x = 3 + 4j

y = 2 - 3j

z = x + y

print(z)
```
```
(5+1j)
```

INTEGER DIVISION

There are two kinds of division operators:

- "true division" performed by "/"
- "floor division" performed by "//"

True division uses the slash (/) character as the operator sign. Most probably it is what you expect "division" to be. The following examples are hopefully self-explanatory:

```python
10 / 3
```
Output: 3.3333333333333335

```python
10.0 / 3.0
```
Output: 3.3333333333333335

```python
10.5 / 3.5
```

Output: `3.0`

The operator "//" performs floor division, i.e., the dividend is divided by the divisor - like in true division - but the floor of the result will be returned. The floor is the largest integer number smaller than the result of the true division. This number will be turned into a float, if either the dividend or the divisor or both are float values. If both are integers, the result will be an integer as well. In other words, "//" always truncates towards negative infinity.

Connection to the floor function: In mathematics and computer science, the floor function is the function that takes as input a real number x and returns the greatest integer floor $(x) = \lfloor x \rfloor$ that is less than or equal to x.

If you are confused now by this rather mathematical and theoretical definition, the following examples will hopefully clarifiy the matter:

```
9 // 3
```
Output: `3`

```
10 // 3
```
Output: `3`

```
11 // 3
```
Output: `3`

```
12 // 3
```
Output: `4`

```
10.0 // 3
```
Output: `3.0`

```
-7 // 3
```
Output: `-3`

```
-7.0 // 3
```
Output: `-3.0`

STRINGS
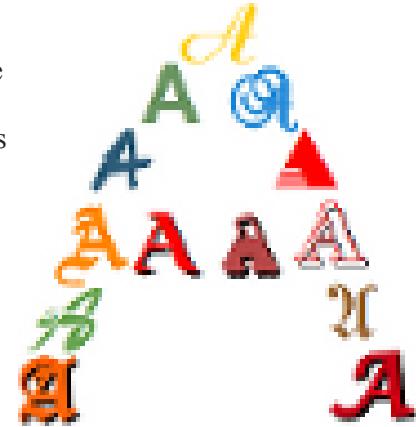
The task of the first-generation computers in the forties and fifties had been - due to technical restraints - focused on number processing. Text processing had been just a dream at that time. Nowadays, one of the main tasks of computers is text processing in all its varieties; the most prominent applications are search engines like Google. To enable text processing programming languages need suitable data types. Strings are used in all modern programming languages to store and process textual information. Logically, a string - like any text - is a sequence of characters. The question remains what a character consists of. In a book, or in a text like the one your are reading now, characters consist of graphical shapes, so-called graphemes, consisting of lines, curves and crossings in certain angles or positions and so on. The ancient Greeks associated the word with the engravings on coins or the stamps on seals.

In computer science or computer technology, a character is a unit of information. These characters correspond to graphemes, the fundamental units of written or printed language. Before Unicode came into usage, there was a one to one relationship between bytes and characters, i.e., every character - of a national variant, i.e. not all the characters of the world - was represented by a single byte. Such a character byte represents the logical concept of this character and the class of graphemes of this character. The image above depicts various representations of the letter "A", i.e., "A" in different fonts. So in printing there are various graphical representations or different "encodings" of the abstract concept of the letter A. (By the way, the letter "A" can be ascribed to an Egyptian hieroglyph with a pictogram of an ox.) All of these graphical representations have certain features in common. In other words, the meaning of a character or a written or printed text doesn't depend on the font or writing style used. On a computer the capital A is encoded in binary form. If we use ASCII it is encoded - like all the other characters - as the byte 65.

ASCII is restricted to 128 characters and "Extended ASCII" is is still limited to 256 bytes or characters. This is good enough for languages like English, German and French, but by far not sufficient for Chinese, Japanese and Korean. That's where Unicode gets into the game. Unicode is a standard designed to represent every character from every language, i.e., it can handle any text of the world's writing systems. These writing systems can also be used simultaneously, i.e., Roman alphabet mixed with Cyrillic or even Chinese characters.

There is a different story about Unicode. A character maps to a code point. A code point is a theoretical concept. That is, for example, that the character "A" is assigned a code point U+0041. The "U+" means "Unicode" and the "0041" is a hexadecimal number, 65 in decimal notation.

You can transform it like this in Python:

```python
hex(65)
```

Output: `'0x41'`

```python
int(0x41)
```

Output: `65`

Up to four bytes are possible per character in Unicode. Theoretically, this means a huge number of 4294967296 possible characters. Due to restrictions from UTF-16 encoding, there are "only" 1,112,064 characters possible. Unicode version 8.0 has assigned 120,737 characters. This means that there are slightly more than 10 % of all possible characters assigned, in other words, we can still add nearly a million characters to Unicode.

UNICODE ENCODINGS

Name	
UTF-32	It's a one to one encoding, i.e., it takes each Unicode character (a 4-byte number) and stores it in 4 bytes. One advantage of this can find the Nth character of a string in linear time, because the Nth character starts at the 4×Nth byte. A serious disadvantage of to the fact that it needs four bytes
UTF-16	UTF-16 (16-bit Unicode Transformation Format) is a character encoding capable of encoding all 1,112,064 valid code poi encoding is variable-length, as code points are encoded with one or tw
UTF-8	UTF8 is a variable-length encoding system for Unicode, i.e., different characters take up a different number of bytes. ASCII chara byte per character. This means that the first 128 characters UTF-8 are indistinguishable from ASCII. But the so-called "Extend like the Umlaute ä, ö and so on take up two bytes. Chinese characters need three bytes. Finally, the very seldom used characters need four bytes to be encoded in UTF-8. W3Techs (Web Technology Surveys) writes that "UTF-8 is used by 94.3% of all character e

STRING, UNICODE AND PYTHON

After this lengthy but necessary introduction, we finally come to Python and the way it deals with strings. All strings in Python 3 are sequences of "pure" Unicode characters, no specific encoding like UTF-8.

There are different ways to define strings in Python:

```
s = 'I am a string enclosed in single quotes.'
s2 = "I am another string, but I am enclosed in double quotes."
```

Both s and s2 of the previous example are variables referencing string objects. We can see that string literals can either be enclosed in matching single (') or in double quotes ("). Single quotes will have to be escaped with a backslash \ , if the string is defined with single quotes:

```
s3 = 'It doesn\'t matter!'
```

This is not necessary, if the string is represented by double quotes:

```
s3 = "It doesn't matter!"
```

Analogously, we will have to escape a double quote inside a double quoted string:
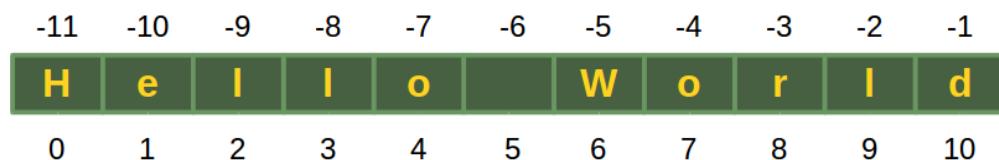
```
txt = "He said: \"It doesn't matter, if you enclose a string in si
ngle or double quotes!\""
print(txt)
```

```
He said: "It doesn't matter, if you enclose a string in single or
double quotes!"
```

They can also be enclosed in matching groups of three single or double quotes. In this case they are called triple-quoted strings. The backslash () character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

```
txt = '''A string in triple quotes can extend
over multiple lines like this one, and can contain
'single' and "double" quotes.'''
```

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A "quote" is the character used to open the string, i.e., either ' or ".)

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

A string in Python consists of a series or sequence of characters - letters, numbers, and special characters. Strings can be subscripted or indexed. Similar to C, the first character of a string has the index 0.

```
s = "Hello World"
s[0]
```

Output: `'H'`

```
s[5]
```

Output: `' '`

The last character of a string can be accessed this way:

```
s[len(s)-1]
```

Output: `'d'`

Yet, there is an easier way in Python. The last character can be accessed with -1, the second to last with -2 and so on:

```
s[-1]
```

Output: `'d'`
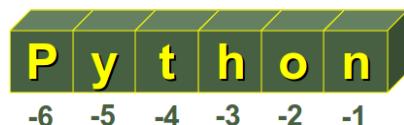
```
s[-2]
```

Output: `'l'`

Some readers might find it confusing when we use "to subscript" as a synonym for "to index". Usually, a subscript is a number, figure, symbol, or indicator that is smaller than the normal line of type and is set slightly below or above it. When we wrote s[0] or s[3] in the previous examples, this can be seen as an alternative way for the notation s0 or s3. So, both s3 and s[3] describe or denote the 4th character. By the way, there is no character type in Python. A character is simply a string of size one.

It's possible to start counting the indices from the right, as we have mentioned previously. In this case negative numbers are used, starting with -1 for the most right character.

- Concatenation

Strings can be glued together (concatenated) with the + operator: "Hello" + "World" will result in "HelloWorld"

- Repetition

String can be repeated or repeatedly concatenated with the asterisk operator "*": "-" * 3 will result in "---"

- Indexing

"Python"[0] will result in "P"

- Slicing

Substrings can be created with the slice or slicing notation, i.e., two indices in square brackets separated by a colon: "Python"[2:4] will result in "th"

- Size

len("Python") will result in 6

IMMUTABLE STRINGS

Like strings in Java and unlike C or C++, Python strings cannot be changed. Trying to change an indexed position will raise an error:

```
s = "Some things are immutable!"
s[-1] = "."
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-53-2fa9c6f1b317> in <module>
      1 s = "Some things are immutable!"
----> 2 s[-1] = "."

TypeError: 'str' object does not support item assignment
```

Beginners in Python are often confused, when they see the following codelines:

```
txt = "He lives in Berlin!"
txt = "He lives in Hamburg!"
```

The variable "txt" is a reference to a string object. We define a completely new string object in the second assignment. So, you shouldn't confuse the variable name with the referenced object!

A STRING PECULIARITY

Strings show a special effect, which we will illustrate in the following example. We will need the "is"-Operator. If both a and b are strings, "a is b" checks if they have the same identity, i.e., share the same memory location. If "a is b" is True, then it trivially follows that "a == b" has to be True as well. Yet, "a == b" True doesn't imply that "a is b" is True as well!

Let's have a look at how strings are stored in Python:

```
a = "Linux"
b = "Linux"
a is b
```
Output: True

Okay, but what happens, if the strings are longer? We use the longest village name in the world in the following example. It's a small village with about 3000 inhabitants in the South of the island of Anglesey in the North-West of Wales:

```
a = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
b = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
a is b
```
Output: True

Nothing has changed in our first "Linux" example. But what works for Wales doesn't work e.g., for Baden-Württemberg in Germany:

```
a = "Baden-Württemberg"
b = "Baden-Württemberg"
a is b
```
Output: False

```
a == b
```
Output: True

You are right, it has nothing to do with geographical places. The special character, i.e., the hyphen, is to "blame".

```
a = "Baden!"
b = "Baden!"
a is b
```
Output: False

```
a = "Baden1"
b = "Baden1"
a is b
```
Output: True

ESCAPE SEQUENCES IN STRINGS

To end our coverage of strings in this chapter, we will introduce some escape characters and sequences. The backslash () character is used to escape characters, i.e., to "escape" the special meaning, which this character would otherwise have. Examples for such characters are newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or 'R'; these strings are called raw strings. Raw strings use different rules for interpreting backslash escape sequences.

Escape Sequence	Meaning
\newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace(BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)

Escape Sequence	Meaning
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (Unicode only)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

BYTE STRINGS

Python 3.0 uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. Every string or text in Python 3 is Unicode, but encoded Unicode is represented as binary data. The type used to hold text is str, the type used to hold data is bytes. It's not possible to mix text and data in Python 3; it will raise TypeError. While a string object holds a sequence of characters (in Unicode), a bytes object holds a sequence of bytes, out of the range 0 to 255, representing the ASCII values. Defining bytes objects and casting them into strings:

```python
x = "Hallo"
t = str(x)
u = t.encode("UTF-8")
print(u)
```

```
b'Hallo'
```

ARITHMETIC AND COMPARISON OPERATORS

INTRODUCTION

This chapter covers the various built-in operators, which Python has to offer.

OPERATORS

These operations (operators) can be applied to all numeric types:

Operator	Description	Example
+, -	Addition, Subtraction	10 -3
*, %	Multiplication, Modulo	27 % 7 Result: 6

/	Division
	This operation brings about different results for Python 2.x (like floor division) and Python 3.x
		Python3:
		`10 / 3`
		`3.3333333333333335`
		
		and in Python 2.x:
		`10 / 3`
		`3`

//	Truncation Division (also known as floordivision or floor division)
	The result of this division is the integral part of the result, i.e. the fractional part is truncated, if there is any.
	It works both for integers and floating-point numbers, but there is a difference between the type of the results: If both the dividend and the divisor are integers, the result will also be an integer. If either the divident or the divisor is a float, the result will be the truncated result as a float.
		`10 // 3`
		`3`
		
		If at least one of the operands is a float value, we get a truncated float value as the result.
		
		`10.0 // 3`
		`3.0`
		`>>>`
		
		A note about efficiency:
		The results of int(10 / 3) and 10 // 3 are equal. But the "//" division is more than two times as fast! You can see this here:
		
		`In [9]: %%timeit`

```
for x in range(1, 10
                 0):
    y = int(100 / x)
                  :
100000 loops, best o
f 3: 11.1 µs per loo
                   p

  In [10]: %%timeit
for x in range(1, 10
                 0):
     y = 100 // x
                 :
100000 loops, best o
f 3: 4.48 µs per loo
                   p
```

+x, -x	Unary minus and Unary plus (Algebraic signs)	-3
~x	Bitwise negation	~3 - 4 Result: -8
**	Exponentiation	10 ** 3 Result: 1000
or, and, not	Boolean Or, Boolean And, Boolean Not	(a or b) and c
in	"Element of"	1 in [3, 2, 1]
$<, \leq, >, \geq$, !=, ==	The usual comparison operators	$2 \leq 3$
|, &, ^	Bitwise Or, Bitwise And, Bitwise XOR	6 ^ 3
<, >	Shift Operators	6 < 3
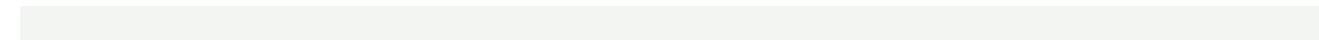
In [ ]:

SEQUENTIAL DATA TYPES

INTRODUCTION

Sequences are one of the principal built-in data types besides numerics, mappings, files, instances and exceptions. Python provides for six sequence (or sequential) data types:

```
strings
byte sequences
byte arrays
lists
tuples
range objects
```

Strings, lists, tuples, bytes and range objects may look like utterly different things, but they still have some underlying concepts in common:

```
The items or elements of strings, lists and tuples are ordered in
a defined sequence
The elements can be accessed via indices
```

```python
text = "Lists and Strings can be accessed via indices!"
print(text[0], text[10], text[-1])
```

```
L S !
```

Accessing lists:

```python
cities = ["Vienna", "London", "Paris",
          "Berlin", "Zurich", "Hamburg"]
print(cities[0])
print(cities[2])
print(cities[-1])
```

```
Vienna
Paris
Hamburg
```

Unlike other programming languages Python uses the same syntax and function names to work on sequential data types. For example, the length of a string, a list, and a tuple can be determined with a function called len():

```
countries = ["Germany", "Switzerland", "Austria",
             "France", "Belgium", "Netherlands",
             "England"]
len(countries)  # the length of the list, i.e. the number of objec
ts
```

Output: 7

```
fib = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
len(fib)
```

Output: 10

BYTES

The byte object is a sequence of small integers. The elements of a byte object are in the range 0 to 255, corresponding to ASCII characters and they are printed as such.

```
s = "Glückliche Fügung"
s_bytes = s.encode('utf-8')
s_bytes
```

Output: b'Gl\xc3\xbcckliche F\xc3\xbcgung'

PYTHON LISTS

So far we had already used some lists, and here comes a proper introduction. Lists are related to arrays of programming languages like C, C++ or Java, but Python lists are by far more flexible and powerful than "classical" arrays. For example, not all the items in a list need to have the same type. Furthermore, lists can grow in a program run, while in C the size of an array has to be fixed at compile time.

Generally speaking a list is a collection of objects. To be more precise: A list in Python is an ordered group of items or elements. It's important to notice that these list elements don't have to be of the same type. It can be an arbitrary mixture of elements like numbers, strings, other lists and so on. The list type is essential for Python scripts and programs, in other words, you will hardly find any serious Python code without a list.

The main properties of Python lists:

- They are ordered
- The contain arbitrary objects
- Elements of a list can be accessed by an index
- They are arbitrarily nestable, i.e. they can contain other lists as sublists
- Variable size
- They are mutable, i.e. the elements of a list can be changed

LIST NOTATION AND EXAMPLES

List objects are enclosed by square brackets and separated by commas. The following table contains some examples of lists:

List	Description
[]	An empty list
[1,1,2,3,5,8]	A list of integers
[42, "What's the question?", 3.1415]	A list of mixed data types
["Stuttgart", "Freiburg", "München", "Nürnberg", "Würzburg", "Ulm","Friedrichshafen", Zürich", "Wien"]	A list of Strings
[["London","England", 7556900], ["Paris","France",2193031], ["Bern", "Switzerland", 123466]]	A nested list
["High up", ["further down", ["and down", ["deep down", "the answer", 42]]]]	A deeply nested list

Assigning a list to a variable:

In [ ]:

```
languages = ["Python", "C", "C++", "Java", "Perl"]
```

There are different ways of accessing the elements of a list. Most probably the easiest way for C programmers will be through indices, i.e. the numbers of the lists are enumerated starting with 0:

```
languages = ["Python", "C", "C++", "Java", "Perl"]
print(languages[0] + " and " + languages[1] + " are quite differen
t!")
print("Accessing the last element of the list: " + languages[-1])
```

```
Python and C are quite different!
Accessing the last element of the list: Perl
```

The previous example of a list has been a list with elements of equal data types. But as we saw before, lists can have various data types, as shown in the following example:

```
group = ["Bob", 23, "George", 72, "Myriam", 29]
```

SUBLISTS

Lists can have sublists as elements. These sublists may contain sublists as well, i.e. lists can be recursively constructed by sublist structures.

```
person = [["Marc", "Mayer"],
          ["17, Oxford Str", "12345", "London"],
          "07876-7876"]
name = person[0]
print(name)
```

```
['Marc', 'Mayer']
```

```
first_name = person[0][0]
print(first_name)
```

```
Marc
```

```
last_name = person[0][1]
print(last_name)
```

```
Mayer
```

```
address = person[1]
print(address)
```

```
['17, Oxford Str', '12345', 'London']
```

```
street = person[1][0]
print(street)
```

```
17, Oxford Str
```

We could have used our variable `address` as well:

```
street = address[0]
print(street)
```

```
17, Oxford Str
```

The next example shows a more complex list with a deeply structured list:

```
complex_list = [["a", ["b", ["c", "x"]]]]
complex_list = [["a", ["b", ["c", "x"]]], 42]
complex_list[0][1]
```

Output: `['b', ['c', 'x']]`

```
complex_list[0][1][1][0]
```

Output: `'c'`

CHANGING LIST

```
languages = ["Python", "C", "C++", "Java", "Perl"]
languages[4] = "Lisp"
languages
```

Output: `['Python', 'C', 'C++', 'Java', 'Lisp']`

```
languages.append("Haskell")
languages
```

Output: `['Python', 'C', 'C++', 'Java', 'Lisp', 'Haskell']`

```
languages.insert(1, "Perl")
languages
```

Output: `['Python', 'Perl', 'C', 'C++', 'Java', 'Lisp', 'Haskell']`

```
shopping_list = ['milk', 'yoghurt', 'egg', 'butter', 'bread', 'ban
anas']
```

We go to a virtual supermarket. Fetch a cart and start shopping:

```
shopping_list = ['milk', 'yoghurt', 'egg', 'butter', 'bread', 'ban
anas']
cart = []
#  "pop()"" removes the last element of the list and returns it
article = shopping_list.pop()
print(article, shopping_list)
cart.append(article)
print(cart)

# we go on like this:
article = shopping_list.pop()
```

```
print("shopping_list:", shopping_list)
cart.append(article)
print("cart: ", cart)
```

```
bananas ['milk', 'yoghurt', 'egg', 'butter', 'bread']
['bananas']
shopping_list: ['milk', 'yoghurt', 'egg', 'butter']
cart:  ['bananas', 'bread']
```

With a while loop:

```
shopping_list = ['milk', 'yoghurt', 'egg', 'butter', 'bread', 'ban
anas']
cart = []
while shopping_list != []:
    article = shopping_list.pop()
    cart.append(article)
    print(article, shopping_list, cart)

print("shopping_list: ", shopping_list)
print("cart: ", cart)
```

```
bananas ['milk', 'yoghurt', 'egg', 'butter', 'bread'] ['bananas']
bread ['milk', 'yoghurt', 'egg', 'butter'] ['bananas', 'bread']
butter ['milk', 'yoghurt', 'egg'] ['bananas', 'bread', 'butter']
egg ['milk', 'yoghurt'] ['bananas', 'bread', 'butter', 'egg']
yoghurt ['milk'] ['bananas', 'bread', 'butter', 'egg', 'yoghurt']
milk [] ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'milk']
shopping_list:  []
cart:  ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'milk']
```

TUPLES

A tuple is an immutable list, i.e. a tuple cannot be changed in any way, once it has been created. A tuple is defined analogously to lists, except the set of elements is enclosed in parentheses instead of square brackets. The rules for indices are the same as for lists. Once a tuple has been created, you can't add elements to a tuple or remove elements from a tuple.

Where is the benefit of tuples?

```
•Tuples are faster than lists.
•If you know that some data doesn't have to be changed, you should
 use tuples instead of lists, because this protects your data agains
 t accidental changes.
•The main advantage of tuples is that tuples can be used as keys i
 n dictionaries, while lists can't.
```

The following example shows how to define a tuple and how to access a tuple. Furthermore, we can see that we raise an error, if we try to assign a new value to an element of a tuple:

```
t = ("tuples", "are", "immutable")
t[0]
```

Output: `'tuples'`

```
t[0] = "assignments to elements are not possible"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-fa4ace6c4d57> in <module>
----> 1 t[0]="assignments to elements are not possible"

TypeError: 'tuple' object does not support item assignment
```

SLICING

In many programming languages it can be quite tough to slice a part of a string and even harder, if you like to address a "subarray". Python makes it very easy with its slice operator. Slicing is often implemented in other languages as functions with possible names like "substring", "gstr" or "substr".

So every time you want to extract part of a string or a list in Python, you should use the slice operator. The syntax is simple. Actually it looks a little bit like accessing a single element with an index, but instead of just one number, we have more, separated with a colon ":". We have a start and an end index, one or both of them may be missing. It's best to study the mode of operation of slice by having a look at examples:

```
slogan = "Python is great"
first_six = slogan[0:6]
first_six
```

Output: `'Python'`

```
starting_at_five = slogan[5:]
starting_at_five
```

Output: `'n is great'`

```
a_copy = slogan[:]
without_last_five = slogan[0:-5]
without_last_five
```

Output: `'Python is '`

Syntactically, there is no difference on lists. We will return to our previous example with European city names:

```
cities = ["Vienna", "London", "Paris", "Berlin", "Zurich", "Hambur
g"]
first_three = cities[0:3]
# or easier:
...
first_three = cities[:3]
print(first_three)
```

```
['Vienna', 'London', 'Paris']
```

Now we extract all cities except the last two, i.e. "Zurich" and "Hamburg":

```
all_but_last_two = cities[:-2]
print(all_but_last_two)
```

```
['Vienna', 'London', 'Paris', 'Berlin']
```

Slicing works with three arguments as well. If the third argument is for example 3, only every third element of the list, string or tuple from the range of the first two arguments will be taken.

If s is a sequential data type, it works like this:

```
    s[begin: end: step]
```

The resulting sequence consists of the following elements:

```
    s[begin], s[begin + 1 * step], ... s[begin + i * step] for all (be
    gin + i * step) < end.
```

In the following example we define a string and we print every third character of this string:

```
slogan = "Python under Linux is great"
slogan[::3]
```

Output: `'Ph d n  e'`

The following string, which looks like a letter salad, contains two sentences. One of them contains covert advertising of my Python courses in Canada:

```
   "TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCda
   ennasdeao"
```

Try to find out the hidden sentence using Slicing. The solution consists of 2 steps!

```
s = "TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB
oCdaennasdeao"
print(s)
```

```
TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCdae
nnasdeao
```

```
s[::2]
```

Output: `'Toronto is the largest City in Canada'`

```
s[1::2]
```

Output: `'Python courses in Toronto by Bodenseo'`

You may be interested in how we created the string. You have to understand list comprehension to understand the following:

```
s = "Toronto is the largest City in Canada"
t = "Python courses in Toronto by Bodenseo"
s = "".join(["".join(x) for x in zip(s,t)])
s
```

Output: `'TPoyrtohnotno ciosu rtshees lianr gTeosrto nCtiot yb yi n
      B oCdaennasdeao'`

LENGTH

Length of a sequence, i.e. a list, a string or a tuple, can be determined with the function len(). For strings it counts the number of characters, and for lists or tuples the number of elements are counted, whereas a sublist counts as one element.

```
txt = "Hello World"
len(txt)
```

Output: 11

```
a = ["Swen", 45, 3.54, "Basel"]
len(a)
```

Output: 4

CONCATENATION OF SEQUENCES

Combining two sequences like strings or lists is as easy as adding two numbers together. Even the operator sign is the same. The following example shows how to concatenate two strings into one:

```python
firstname = "Homer"
surname = "Simpson"
name = firstname + " " + surname
print(name)
```

```
Homer Simpson
```

It's as simple for lists:

```python
colours1 = ["red", "green","blue"]
colours2 = ["black", "white"]
colours = colours1 + colours2
print(colours)
```

```
['red', 'green', 'blue', 'black', 'white']
```

The augmented assignment "+=" which is well known for arithmetic assignments work for sequences as well.

s += t

is syntactically the same as:

s = s + t

But it is only syntactically the same. The implementation is different: In the first case the left side has to be evaluated only once. Augment assignments may be applied for mutable objects as an optimization.

CHECKING IF AN ELEMENT IS CONTAINED IN LIST

It's easy to check, if an item is contained in a sequence. We can use the "in" or the "not in" operator for this purpose. The following example shows how this operator can be applied:

```python
abc = ["a","b","c","d","e"]
"a" in abc
```

Output: True

```python
"a" not in abc
```

Output: `False`

```
"e" not in abc
```
Output: `False`

```
"f" not in abc
```
Output: `True`

```
slogan = "Python is easy!"
"y" in slogan
```
Output: `True`

```
"x" in slogan
```
Output: `False`

REPETITIONS

So far we had a " + " operator for sequences. There is a " * " operator available as well. Of course, there is no "multiplication" between two sequences possible. " * " is defined for a sequence and an integer, i.e. "s * n" or "n * s". It's a kind of abbreviation for an n-times concatenation, i.e.

str * 4

is the same as

str + str + str + str

Further examples:

```
3 * "xyz-"
```
Output: `'xyz-xyz-xyz-'`

```
"xyz-" * 3
```
Output: `'xyz-xyz-xyz-'`

```
3 * ["a","b","c"]
```
Output: `['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']`

The augmented assignment for "*" *can be used as well: s* = n is the same as s = s * n.

THE PITFALLS OF REPETITIONS
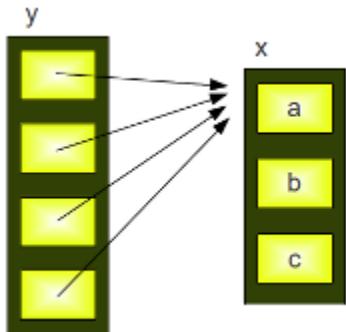
In our previous examples we applied the repetition operator on strings and flat lists. We can apply it to nested lists as well:

```
x = ["a","b","c"]
y = [x] * 4
y
```

Output:
```
[['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c'], ['a',
 'b', 'c']]
```

```
y[0][0] = "p"
y
```

Output:
```
[['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c'], ['p',
 'b', 'c']]
```


This result is quite astonishing for beginners of Python programming. We have assigned a new value to the first element of the first sublist of y, i.e. y[0][0] and we have "automatically" changed the first elements of all the sublists in y, i.e. y[1][0], y[2][0], y[3][0].

The reason is that the repetition operator "* 4" creates 4 references to the list x: and so it's clear that every element of y is changed, if we apply a new value to y[0][0].

LISTS

CHANGING LISTS

This chapter of our tutorial deals with further aspects of lists. You will learn how to append and insert objects to lists and you will also learn how to delete and remove elements by using 'remove' and 'pop'.

A list can be seen as a stack. A stack in computer science is a data structure, which has at least two operations: one which can be used to put or push data on the stack, and another one to take away the most upper element of the stack. The way of working can be imagined with a stack of plates. If you need a plate you will usually take the most upper one. The used plates will be put back on the top of the stack after cleaning. If a programming language supports a stack like data structure, it will also supply at least two operations:

```
•push
This method is used to put a new object on the
stack. Depending on the point of view, we say t
hat we "push" the object on top or attach it t
o the right side. Python doesn't offer - contra
ry to other programming languages - no method with the name "pus
h", but the method "append" has the same functionality.
•pop
This method returns the top element of the stack. The object will b
e removed from the stack as well.
•peek
Some programming languages provide another method, which can be use
d to view what is on the top of the stack without removing this ele
ment. The Python list class doesn't possess such a method, because
it is not needed. A peek can be simulated by accessing the element
with the index -1:
```

```
lst = ["easy", "simple", "cheap", "free"]
lst[-1]
```
Output: `'free'`

POP AND APPEND

•lst.append(x)

This method appends an element to the end of the list "lst".

```
lst = [3, 5, 7]
```

```
lst.append(42)
lst
```

Output: `[3, 5, 7, 42]`

It's important to understand that append returns "None". In other words, it usually doesn't make sense to reassign the return value:

```
lst = [3, 5, 7]
lst = lst.append(42)
print(lst)
```

```
None
```

• lst.pop(i)

'pop' returns the (i)th element of a list "lst". The element will be removed from the list as well.

```
cities = ["Hamburg", "Linz", "Salzburg", "Vienna"]
cities.pop(0)
```

Output: `'Hamburg'`

```
cities
```

Output: `['Linz', 'Salzburg', 'Vienna']`

```
cities.pop(1)
```

Output: `'Salzburg'`

```
cities
```

Output: `['Linz', 'Vienna']`

The method 'pop' raises an IndexError exception, if the list is empty or the index is out of range.

• s.pop()

The method 'pop' can be called without an argument. In this case, the last element will be returned. So s.pop() is equivalent to s.pop(-1).

```
cities = ["Amsterdam", "The Hague", "Strasbourg"]
cities.pop()
```

Output: `'Strasbourg'`

```
cities
```

`['Amsterdam', 'The Hague']`

EXTEND

We saw that it is easy to append an object to a list. What about adding more than one element to a list? Maybe, you want to add all the elements of another list to your list. If you use append, the other list will be appended as a sublist, as we can see in the following example:

```
lst = [42,98,77]
lst2 = [8,69]
lst.append(lst2)
lst
```

Output: `[42, 98, 77, [8, 69]]`

For this purpose, Python provides the method 'extend'. It extends a list by appending all the elements of an iterable like a list, a tuple or a string to a list:

```
lst = [42,98,77]
lst2 = [8,69]
lst.extend(lst2)
lst
```

Output: `[42, 98, 77, 8, 69]`

As we have mentioned, the argument of 'extend' doesn't have to be a list. It can be any kind of iterable. That is, we can use tuples and strings as well:

```
lst = ["a", "b", "c"]
programming_language = "Python"
lst.extend(programming_language)
print(lst)
```

```
['a', 'b', 'c', 'P', 'y', 't', 'h', 'o', 'n']
```

Now with a tuple:

```
lst = ["Java", "C", "PHP"]
t = ("C#", "Jython", "Python", "IronPython")
lst.extend(t)
lst
```

Output: `['Java', 'C', 'PHP', 'C#', 'Jython', 'Python', 'IronPython']`

EXTENDING AND APPENDING LISTS WITH THE '+' OPERATOR

There is an alternative to 'append' and 'extend'. '+' can be used to combine lists.

```
level = ["beginner", "intermediate", "advanced"]
other_words = ["novice", "expert"]
level + other_words
```

Output: `['beginner', 'intermediate', 'advanced', 'novice', 'expert']`

Be careful. Never ever do the following:

```
L = [3, 4]
L = L + [42]
L
```

Output: `[3, 4, 42]`

Even though we get the same result, it is not an alternative to 'append' and 'extend':

```
L = [3, 4]
L.append(42)
L
```

Output: `[3, 4, 42]`

```
L = [3, 4]
L.extend([42])
L
```

Output: `[3, 4, 42]`

The augmented assignment (+=) is an alternative:

```
L = [3, 4]
L += [42]
L
```

Output: `[3, 4, 42]`

In the following example, we will compare the different approaches and calculate their run times. To understand the following program, you need to know that time.time() returns a float number, the time in seconds since the so-called „The Epoch"1. time.time() - start_time calculates the time in seconds used for the for loops:

```
import time

n= 100000

start_time = time.time()
l = []
for i in range(n):
    l = l + [i * 2]
print(time.time() - start_time)
```

29.277812480926514

```
start_time = time.time()
l = []
for i in range(n):
    l += [i * 2]
print(time.time() - start_time)
```

0.04687356948852539

```
start_time = time.time()
l = []
for i in range(n):
    l.append(i * 2)
print(time.time() - start_time)
```

0.03689885139465332

This program returns shocking results:

We can see that the "+" operator is about 1268 times slower than the append method. The explanation is easy: If we use the append method, we will simply append a further element to the list in each loop pass. Now we come to the first loop, in which we use l = l + [i * 2]. The list will be copied in every loop pass. The new element will be added to the copy of the list and result will be reassigned to the variable l. After that, the old list will have to be removed by Python, because it is not referenced anymore. We can also see that the version with the augmented assignment ("+="), the loop in the middle, is only slightly slower than the version using "append".

REMOVING AN ELEMENT WITH REMOVE

It is possible to remove a certain value from a list without knowing the position with the method "remove" .

```
s.remove(x)
```

This call will remove the first occurrence of 'x' from the list 's'. If 'x' is not in the list, a ValueError will be raised. We will call the remove method three times in the following example to remove the colour "green". As the colour "green" occurrs only twice in the list, we get a ValueError at the third time:

```
colours = ["red", "green", "blue", "green", "yellow"]
colours.remove("green")
colours
```

Output: `['red', 'blue', 'green', 'yellow']`

```
colours.remove("green")
colours
```

Output: `['red', 'blue', 'yellow']`

```
colours.remove("green")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-278a6b3f3483> in <module>
----> 1 colours.remove("green")

ValueError: list.remove(x): x not in list
```

FIND THE POSITION OF AN ELEMENT IN A LIST

The method "index" can be used to find the position of an element within a list:

```
s.index(x[, i[, j]])
```

It returns the first index of the value x. A ValueError will be raised, if the value is not present. If the optional parameter i is given, the search will start at the index i. If j is also given, the search will stop at position j.

```
colours = ["red", "green", "blue", "green", "yellow"]
colours.index("green")
```

Output: 1

```
colours.index("green", 2)
```

Output: 3

```
colours.index("green", 3,4)
```

Output: 3

```
colours.index("black")
```

```
---------------------------------------------------------------------------
ValueError                                  Traceback (most recent call last)
<ipython-input-30-b6026e7dbb87> in <module>
----> 1 colours.index("black")

ValueError: 'black' is not in list
```

INSERT

We have learned that we can put an element to the end of a list by using the method "append". To work efficiently with a list, we need also a way to add elements to arbitrary positions inside of a list. This can be done with the method "insert":

```
s.insert(index, object)
```

An object "object" will be included in the list "s". "object" will be placed before the element s[index]. s[index] will be "object" and all the other elements will be moved one to the right.

```
lst = ["German is spoken", "in Germany,", "Austria", "Switzerlan
d"]
lst.insert(3, "and")
lst
```

Output:
```
['German is spoken', 'in Germany,', 'Austria', 'and', 'Switze
rland']
```

The functionality of the method "append" can be simulated with insert in the following way:

```
abc = ["a","b","c"]
abc.insert(len(abc),"d")
abc
```

Output:
```
['a', 'b', 'c', 'd']
```

Footnotes:

1 Epoch time (also known as Unix time or POSIX time) is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds.

SHALLOW AND DEEP COPY

INTRODUCTION

In this chapter, we will cover the question of how to copy lists and nested lists. The problems which we will encounter are general problems of mutable data types. Trying to copy lists can be a stumping experience for newbies. But before, we would like to summarize some insights from the previous chapter "Data Types and Variables". Python even shows a strange behaviour for the beginners of the language - in comparison with some other traditional programming languages - when assigning and copying simple data types like integers and strings. The difference between shallow and deep copying is only relevant for compound objects, i.e. objects containing other objects, like lists or class instances.

In the following code snippet, y points to the same memory location as X. We can see this by applying the id() function on x and y. However, unlike "real" pointers like those in C and C++, things change, when we assign a new value to y. In this case y will receive a separate memory location, as we have seen in the chapter "Data Types and Variables" and can see in the following example:

```
x = 3
y = x
print(id(x), id(y))
```

```
94308023129184 94308023129184
```

```
y = 4
print(id(x), id(y))
```

```
94308023129184 94308023129216
```

```
print(x,y)
```

```
3 4
```

But even if this internal behaviour appears strange compared to programming languages like C, C++, Perl or Java, the observable results of the previous assignments are what we expected, i.e. if you do not look at the id values. However, it can be problematic, if we copy mutable objects like lists and dictionaries.

Python creates only real copies, if it has to, i.e. if the user, the programmer, explicitly demands it.

We will introduce you to the most crucial problems, which can occur when copying mutable objects such as lists and dictionaries.

VARIABLES SHARING AN OBJECT

You are on this page to learn about copying objects, especially lists. Nevertheless, you need to exercise patience. We want to show something that looks like a copy to many beginners but has nothing to do with copies.

```
colours1 = ["red", "blue"]
colours2 = colours1
print(colours1, colours2)
```

```
['red', 'blue'] ['red', 'blue']
```

Both variables reference the same list object. If we look at the identities of the variables `colours1` and `colours2`, we can see that both are references to the same object:
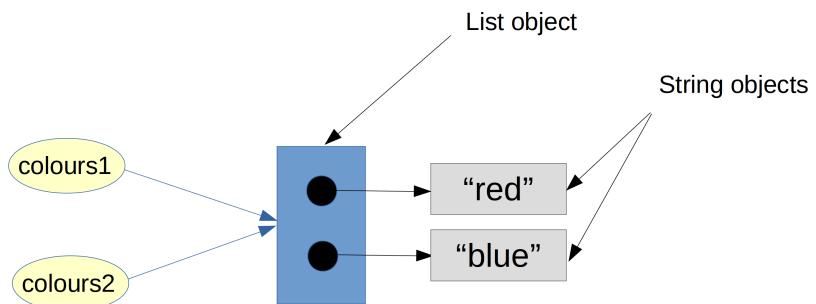
```
print(id(colours1), id(colours2))
```

```
139720054914312 139720054914312
```

In the example above a simple list is assigned to colours1. This list is a so-called "shallow list", because it doesn't have a nested structure, i.e. no sublists are contained in the list. In the next step we assign colours1 to colours2.

The id() function shows us that both variables point at the same list object, i.e. they share this object.

We have two variable names `colours1` and `colours2`, which we have depicted as yellow ovals. The blue box symbolizes the list object. A list object consists of references to other objects. In our example the list object, which is referenced by both variables, references two string objects, i.e. "red" and "blue". Now we have to examine, what will happen, if we change just one element of the list of `colours2` or `colours1`.

Now we want to see, what happens, if we assign a new object to `colours2`. As expected, the values of `colours1` remain unchanged. Like it was in our example in the chapter "Data Types and Variables", a new memory location had been allocated for `colours2`, as we have assigned a completely new list, i.e. a new list object to this variable.

```
colours1 = ["red", "blue"]
colours2 = colours1
```

```python
print(id(colours1), id(colours2))
```

```
139720055355208 139720055355208
```

```python
colours2 = "green"
print(colours1)
```

```
['red', 'blue']
```

```python
print(colours1)
```

```
['red', 'blue']
```

```python
print(colours2)
```
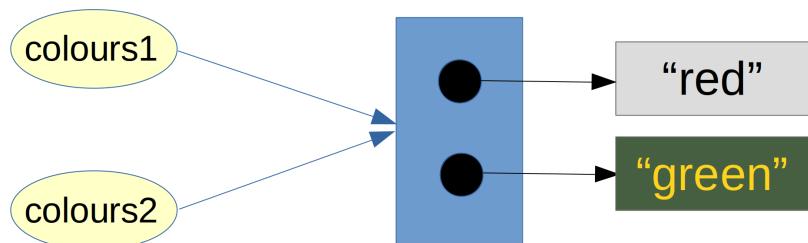
```
green
```

We assigned a new object to the variable 'colours2'. In the following code, we will change the list object internally by assigning a new value to the second element of the list.

```python
colours1 = ["red", "blue"]
colours2 = colours1

colours2[1] = "green"
```

Let's see, what has happened in detail in the previous code. We assigned a new value to the second element of colours2, i.e. the element with the index 1. Lots of beginners will be surprised as the list of colours1 has been "automatically" changed as well. Of course, we don't have two lists: We have only two names for the same list!

The explanation is that we didn't assign a new object to the variable `colours2`. We changed the object referenced by `colours2` internally or as it is usually called "in-place". Both variables `colours1` and `colours2` still point to the same list object.

COPYING LISTS

We have finally arrived at the topic of copying lists. The `list` class offers the method `copy` for this purpose. Even though the name seems to be clear, this path also has a catch.

The catch can be seen, if we use help on `copy`:

```
help(list.copy)
```
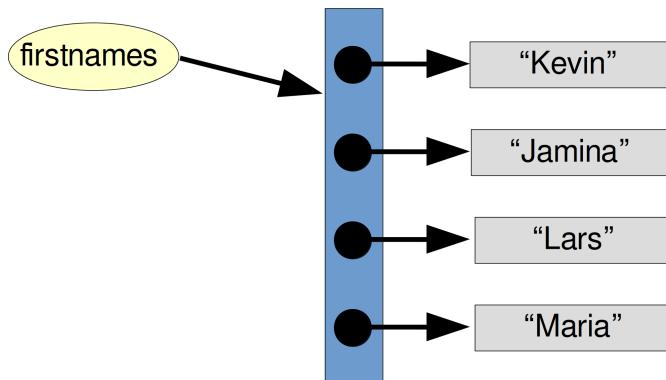
```
Help on method_descriptor:

copy(self, /)
    Return a shallow copy of the list.
```

Many beginners overlook the word "shallow" here. `help` tells us that a new list will be created by the `copy` method. This new list will be a 'shallow' copy of the original list.

In principle, the word "shallow" is unnecessary or even misleading in this definition.
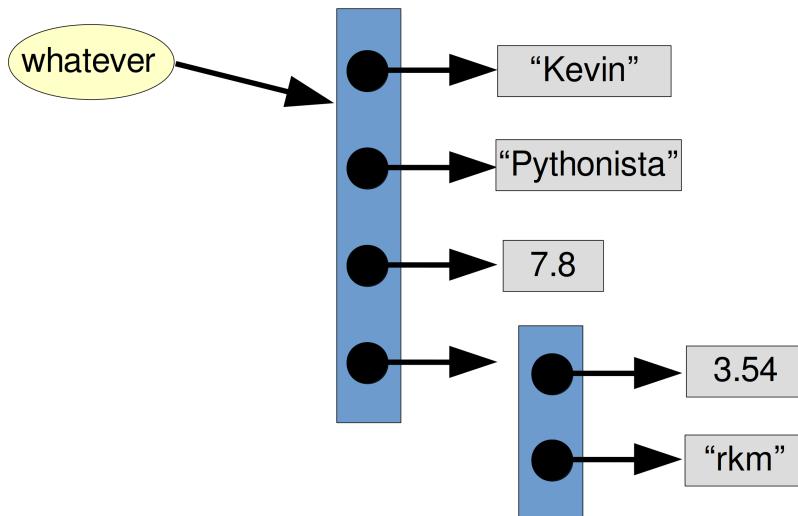
First you should remember what a list is in Python. A list in Python is an object consisting of an ordered sequence of references to Python objects. The following is a list of strings:

The list which the variable `firstnames` is referencing is a list of strings. Basically, the list object is solely the blue box with the arrows, i.e. the references to the strings. The strings itself are not part of the list.

```
firstnames = ['Kevin', 'Jamina', 'Lars', 'Maria']
```

The previous example the list of first names `firstnames` is homogeneous, i.e. it consists only of strings, so all elements have the same data type. But you should be aware of the fact that the references of a list can refer to any objects. The following list `whatever` is such a more general list:

```
whatever = ["Kevin", "Pythonista", 7.8, [3.54, "rkm"]]
```

When a list is copied, we copy the references. In our examples, these are the blue boxes referenced by `firstnames` and by `whatever` . The implications of this are shown in the following subchapter.
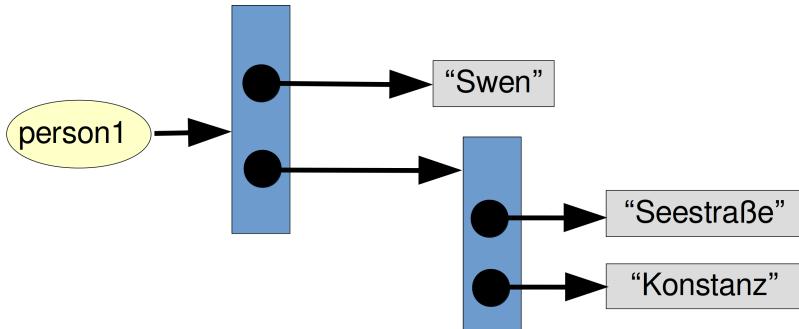
PROBLEMS OF COPYING LISTS

Copying lists can easily be misunderstood, and this misconception can lead to unpleasant errors. We will present this in the form of a slightly different love story.

Imagine a person living in the city of Constance. The city is located in the South of Germany at the banks of Lake Constance (Bodensee in German). The river Rhine, which starts in the Swiss Alps, passes through Lake Constance and leaves it, considerably larger. This person called Swen lives in "Seestrasse", one of the most expensive streets of Constance. His apartment has a direct view of the lake and the city of Constance. We put some information about Swen in the following list structure:

```python
person1 = ["Swen", ["Seestrasse", "Konstanz"]]
```
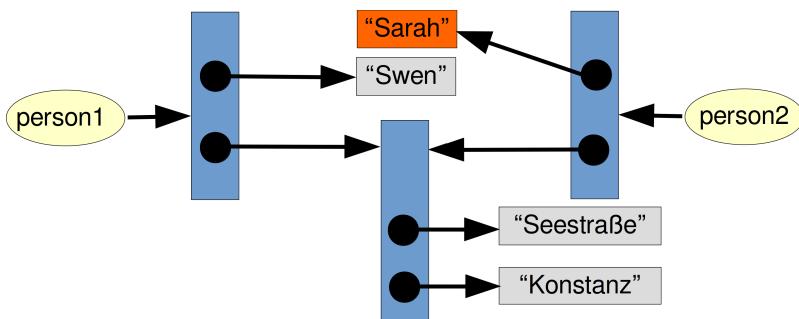

One beautiful day, the good-looking Sarah meets Swen, her Prince Charming. To cut the story short: Love at first sight and she moves in with Swen.

Now it is up to us to create a data set for Sarah. We are lazy and we will recycle Swen's data, because she will live now with him in the same apartment.

We will copy Swen's data and change the name to Sarah:

```python
person2 = person1.copy()
person2[0] = "Sarah"
print(person2)
```

```
['Sarah', ['Seestrasse', 'Konstanz']]
```


They live in perfect harmony and deep love for a long time, let's say a whole weekend. She suddenly realizes that Swen is a monster: He stains the butter with marmelade and honey and even worse, he spreads out his worn socks in the bedroom. It doesn't take long for her to make a crucial decision: She will leave him and the dream apartment.

She moves into a street called Bücklestrasse. A residential area that is nowhere near as nice, but at least she is away from the monster.

How can we arrange this move from the Python point of view?

The street can be accessed with `person2[1][0]`. So we set this to the new location:

```
person2[1][0] = "Bücklestraße"
```
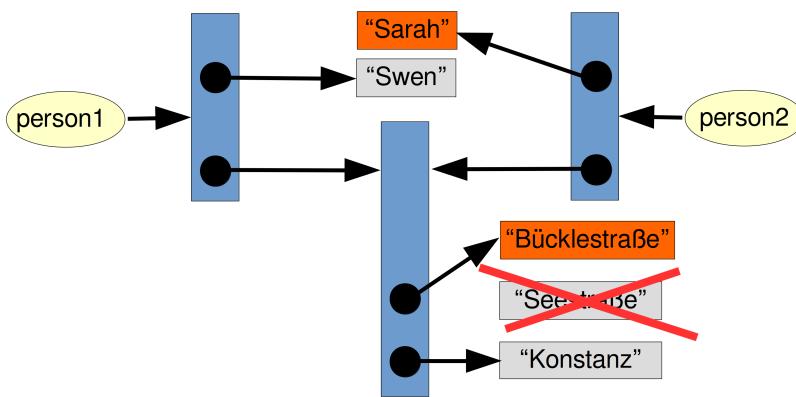
We can see that Sarah successfully moved to the new location:

```
print(person2)
```

```
['Sarah', ['Bücklestraße', 'Konstanz']]
```

Is this the end of our story? Will she never see Swen again? Let's check Swen's data:

```
print(person1)
```

```
['Swen', ['Bücklestraße', 'Konstanz']]
```

Swen is clingy. She cannot get rid of him! This might be quite surprising for some. Why is it like this? The list `person1` consists of two references: One to a string object ("Swen") and the other one to nested list (the address `['Seestrasse', 'Konstanz']`. When we used `copy`, we copied only these references. This means that both `person1[1]` and `person2[1]` reference the same list object. When we change this nested list, it is visible in both lists.

DEEPCOPY FROM THE MODULE COPY
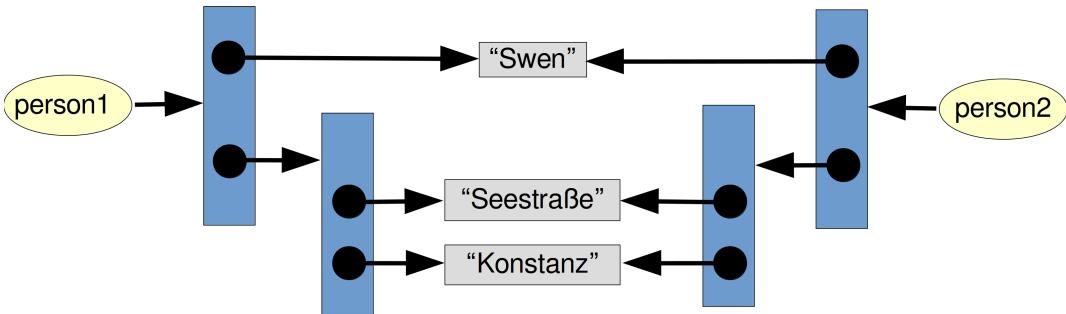
A solution to the described problem is provided by the module `copy`. This module provides the method "deepcopy", which allows a complete or deep copy of an arbitrary list, i.e. shallow and other lists.

Let us redo the previous example with the function `deepcopy`:

```python
from copy import deepcopy
person1 = ["Swen", ["Seestrasse", "Konstanz"]]

person2 = deepcopy(person1)
person2[0] = "Sarah"
```

After this the implementation structure looks like this:
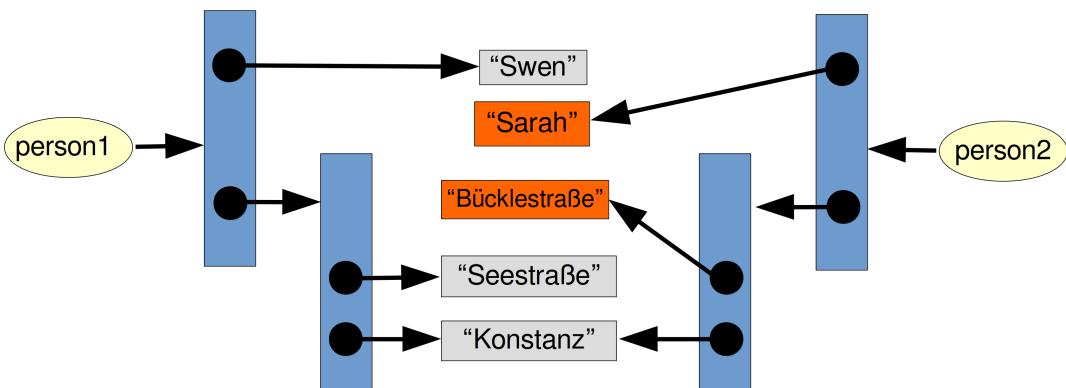
We can see that the nested list with the address was copied as well. We are now well prepared for Sarah's escape-like move.

```python
person2[1][0] = "Bücklestrasse"
```

She successfully moved out and left Swen for good, as we can see in the following:

```python
print(person1, person2)
```

```
['Swen', ['Seestrasse', 'Konstanz']] ['Sarah', ['Bücklestrasse',
'Konstanz']]
```

For the sake of clarity, we also provide a diagram here.

We can see by using the id function that the sublist has been copied, because `id(person1[1])` is different from `id(person2[1])`. An interesting fact is that the strings are not copied: `person1[1]` and `person2[1]` reference the same string.

```
print(id(person1[1]), id(person2[1]))
```

139720051192456 139720054073864

DICTIONARIES

INTRODUCTION

We have already become acquainted with lists in the previous chapter. In this chapter of our online Python course we will present the dictionaries, the operators and the methods on dictionaries. Python programs or scripts without lists and dictionaries are nearly inconceivable. Dictionaries and their powerful implementations are part of what makes Python so effective and superior. Like lists, they can be easily changed, can be shrunk and grown ad libitum at run time. They shrink and grow without the necessity of making copies. Dictionaries can be contained in lists and vice versa.

But what's the difference between lists and dictionaries? A list is an ordered sequence of objects, whereas dictionaries are unordered sets. However, the main difference is that items in dictionaries are accessed via keys and not via their position.

More theoretically, we can say that dictionaries are the Python implementation of an abstract data type, known in computer science as an associative array. Associative arrays consist - like dictionaries of (key, value) pairs, such that each possible key appears at most once in the collection. Any key of the dictionary is associated (or mapped) to a value. The values of a dictionary can be any type of Python data. So, dictionaries are unordered key-value-pairs. Dictionaries are implemented as hash tables, and that is the reason why they are known as "Hashes" in the programming language Perl.

Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists. Dictionaries belong to the built-in mapping type, but so far, they are the sole representative of this kind!

At the end of this chapter, we will demonstrate how a dictionary can be turned into one list, containing (key,value)-tuples or two lists, i.e. one with the keys and one with the values. This transformation can be done reversely as well.

EXAMPLES OF DICTIONARIES

Our first example is a dictionary with cities located in the US and Canada and their corresponding population. We have taken those numbers out the "List of North American cities by population" from Wikipedia (https://en.wikipedia.org/wiki/List_of_North_American_cities_by_population)

If we want to get the population of one of those cities, all we have to do is to use the name of the city as an index. We can see that dictonaries are enclosed in curly brackets. They contain key value pairs. A key and its corresponding value are separated by a colon:

```python
city_population = {"New York City": 8_550_405,
                   "Los Angeles": 3_971_883,
                   "Toronto": 2_731_571,
                   "Chicago": 2_720_546,
```

```
                    "Houston": 2_296_224,
                    "Montreal": 1_704_694,
                    "Calgary": 1_239_220,
                    "Vancouver": 631_486,
                    "Boston": 667_137}
```

Output: 8550405

We can access the value for a specific key by putting this key in brackets following the name of the dictionary:

```
city_population["New York City"]
```

Output: 8550405

```
city_population["Toronto"]
```

Output: 2731571

```
city_population["Boston"]
```

Output: 667137

What happens, if we try to access a key, i.e. a city in our example, which is not contained in the dictionary? We raise a KeyError:

```
city_population["Detroit"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-80e422418d76> in <module>
----> 1 city_population["Detroit"]

KeyError: 'Detroit'
```

A frequently asked question is if dictionary objects are ordered. The uncertainty arises from the fact that dictionaries were not sorted in versions before Python 3.7. In Python 3.7 and all later versions, dictionaries are sorted by the order of item insertion. In our example this means the dictionary keeps the order in which we defined the dictionary. You can see this by printing the dictionary:

```
city_population
```

{'New York City': 8550405,
       'Los Angeles': 3971883,
       'Toronto': 2731571,
       'Chicago': 2720546,
       'Houston': 2296224,
       'Montreal': 1704694,
       'Calgary': 1239220,
       'Vancouver': 631486,
       'Boston': 667137}

Yet, ordering doesn't mean that you have a way of directly calling the nth element of a dictionary. So trying to access a dictionary with a number - like we do with lists - will result in an exception:

```
city_population[0]
```

```
-------------------------------------------------------------------
KeyError                                    Traceback (most recent call last)
<ipython-input-7-a4816f909f86> in <module>
----> 1 city_population[0]

KeyError: 0
```

It is very easy to add another entry to an existing dictionary:

```
city_population["Halifax"] = 390096
city_population
```

Output: {'New York City': 8550405,
       'Los Angeles': 3971883,
       'Toronto': 2731571,
       'Chicago': 2720546,
       'Houston': 2296224,
       'Montreal': 1704694,
       'Calgary': 1239220,
       'Vancouver': 631486,
       'Boston': 667137,
       'Halifax': 390096}

So, it's possible to create a dictionary incrementally by starting with an empty dictionary. We haven't mentioned so far, how to define an empty one. It can be done by using an empty pair of brackets. The following defines an empty dictionary called city:

In [ ]:
```
city_population = {}
city_population
```

```
city_population['New York City'] = 8550405
city_population['Los Angeles'] = 3971883
city_population
```

Looking at our first examples with the cities and their population, you might have gotten the wrong impression that the values in the dictionaries have to be different. The values can be the same, as you can see in the following example. In honour to the patron saint of Python "Monty Python", we'll have now some special food dictionaries. What's Python without "bacon", "egg" and "spam"?

In [ ]:
```
food = {"bacon": "yes", "egg": "yes", "spam": "no" }
food
```

Keys of a dictionary are unique. In casse a keys is defined multiple times, the value of the last "wins":

In [ ]:
```
food = {"bacon" : "yes", "spam" : "yes", "egg" : "yes", "spam" :
"no" }
food
```

Our next example is a simple English-German dictionary:

In [ ]:
```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yello
w":"gelb"}
print(en_de)
print(en_de["red"])
```

What about having another language dictionary, let's say German-French?

Now it's even possible to translate from English to French, even though we don't have an English-French-dictionary. de_fr[en_de["red"]] gives us the French word for "red", i.e. "rouge":

In [ ]:
```
de_fr = {"rot": "rouge", "grün": "vert", "blau": "bleu", "gelb":
"jaune"}
en_de = {"red": "rot", "green": "grün", "blue": "blau", "yellow":
"gelb"}
print(en_de)
```

In [ ]:
```
print(en_de["red"])
```

In [ ]:
```
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gel
b":"jaune"}
print("The French word for red is: " + de_fr[en_de["red"]])
```

We can use arbitrary types as values in a dictionary, but there is a restriction for the keys. Only immutable data types can be used as keys, i.e. no lists or dictionaries can be used: If you use a mutable data type as a key, you get an error message:

In [ ]:
```
dic = {[1,2,3]: "abc"}
```

Tuple as keys are okay, as you can see in the following example:

In [ ]:
```
dic = {(1, 2, 3): "abc", 3.1415: "abc"}
dic
```

Let's improve our examples with the natural language dictionaries a bit. We create a dictionary of dictionaries:

In [ ]:
```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yello
w":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gel
b":"jaune"}
de_tr = {"rot": "kırmızı", "grün": "yeşil", "blau": "mavi", "gel
b": "jel"}
en_es = {"red" : "rojo", "green" : "verde", "blue" : "azul", "yell
ow":"amarillo"}

dictionaries = {"en_de" : en_de, "de_fr" : de_fr, "de_tr": de_tr,
"en_es": en_es}
dictionaries
```

In [ ]:
```
cn_de = {"红": "rot", "绿" : "grün", "蓝" : "blau", "黄" : "gelb"}
de_ro = {'rot': 'roşu', 'gelb': 'galben', 'blau': 'albastru', 'grü
n': 'verde'}
de_hex = {"rot" : "#FF0000", "grün" : "#00FF00", "blau" : "0000F
F", "gelb":"FFFF00"}
en_pl = {"red" : "czerwony", "green" : "zielony",
         "blue" : "niebieski", "yellow" : "żółty"}
de_it = {"rot": "rosso", "gelb": "giallo", "blau": "blu", "grün":
"verde"}
```

```
dictionaries["cn_de"] = cn_de
dictionaries["de_ro"] = de_ro
dictionaries["de_hex"] = de_hex
dictionaries["en_pl"] = en_pl
dictionaries["de_it"] = de_it
dictionaries
```

A dictionary of dictionaries.

In [ ]:
```
dictionaries["en_de"]      # English to German dictionary
```

In [ ]:
```
dictionaries["de_fr"]      # German to French
```

In [ ]:
```
print(dictionaries["de_fr"]["blau"])     # equivalent to de_fr['bla
u']
```

In [ ]:
```
de_fr['blau']
```

In [ ]:
```
lang_pair = input("Which dictionary, e.g. 'de_fr', 'en_de': ")
word_to_be_translated = input("Which colour: ")

d = dictionaries[lang_pair]
if word_to_be_translated in d:
    print(word_to_be_translated + " --> " + d[word_to_be_translate
d])
```

In [ ]:
```
dictionaries['de_fr'][dictionaries['en_de']['red']]
```

In [ ]:
```
de_fr
```

In [ ]:
```
for value in de_fr.values():
    print(value)
```

In [ ]:
```
for key, value in de_fr.items():
    print(key, value)
```

In [ ]:

```
fr_de = {}
fr_de['rouge'] = 'rot'
fr_de['vert'] = "grün"
```

In [ ]:

```
fr_de = {}
for key, value in de_fr.items():
    fr_de[value] = key              # key and value are swapped

fr_de
```

In [ ]:

```
de_cn = {}
for key, value in cn_de.items():
    de_cn[value] = key
de_cn
```

OPERATORS IN DICTIONARIES

Operator	Explanation
len(d)	returns the number of stored entries, i.e. the number of (key,value) pairs.
del d[k]	deletes the key k together with his value
k in d	True, if a key k exists in the dictionary d
k not in d	True, if a key k doesn't exist in the dictionary d

Examples: The following dictionary contains a mapping from latin characters to morsecode.

In [ ]:

```
morse = {
"A" : ".-",
"B" : "-...",
"C" : "-.-.",
"D" : "-..",
"E" : ".",
"F" : "..-.",
"G" : "--.",
"H" : "....",
"I" : "..",
"J" : ".---",
```

```
"K" : "-.-",
"L" : ".-..",
"M" : "--",
"N" : "-.",
"O" : "---",
"P" : ".--.",
"Q" : "--.-",
"R" : ".-.",
"S" : "...",
"T" : "-",
"U" : "..-",
"V" : "...-",
"W" : ".--",
"X" : "-..-",
"Y" : "-.--",
"Z" : "--..",
"0" : "-----",
"1" : ".----",
"2" : "..---",
"3" : "...--",
"4" : "....-",
"5" : ".....",
"6" : "-....",
"7" : "--...",
"8" : "---..",
"9" : "----.",
"." : ".-.-.-",
"," : "--..--"
}
```

If you save this dictionary as morsecode.py, you can easily follow the following examples. At first you have to import this dictionary:

In [ ]:

```
from morsecode import morse
```

The numbers of characters contained in this dictionary can be determined by calling the len function:

In [ ]:
```
len(morse)
38
```

The dictionary contains only upper case characters, so that "a" returns False, for example:

```
In [ ]:
"a" in morse
```

```
In [ ]:
"A" in morse
```

```
In [ ]:
"a" not in morse
```

```
In [ ]:
word = input("Your word: ")

for char in word.upper():
    print(char, morse[char])
```

```
In [ ]:
word = input("Your word: ")

morse_word = ""
for char in word.upper():
    if char == " ":
        morse_word += "   "
    else:
        if char not in morse:
            continue           # continue with next char, go back t
o for
        morse_word += morse[char] + " "

print(morse_word)
```

```
In [ ]:
for i in range(15):
    if i == 13:
        continue
    print(i)
```

POP() AND POPITEM()

POP

Lists can be used as stacks and the operator pop() is used to take an element from the stack. So far, so good for lists, but does it make sense to have a pop() method for dictionaries? After all, a dict is not a sequence data type, i.e. there is no ordering and no indexing. Therefore, pop() is defined differently with dictionaries. Keys and values are implemented in an arbitrary order, which is not random, but depends on the implementation. If D is a dictionary, then D.pop(k) removes the key k with its value from the dictionary D and returns the

corresponding value as the return value, i.e. D[k].

In [ ]:

```
en_de = {"Austria":"Vienna", "Switzerland":"Bern", "Germany":"Berl
in", "Netherlands":"Amsterdam"}
capitals = {"Austria":"Vienna", "Germany":"Berlin", "Netherland
s":"Amsterdam"}
capital = capitals.pop("Austria")
print(capital)
```

If the key is not found, a KeyError is raised:

In [ ]:

```
print(capitals)
{'Netherlands': 'Amsterdam', 'Germany': 'Berlin'}
capital = capitals.pop("Switzerland")
```

If we try to find out the capital of Switzerland in the previous example, we raise a KeyError. To prevent these errors, there is an elegant way. The method pop() has an optional second parameter, which can be used as a default value:

In [ ]:

```
capital = capitals.pop("Switzerland", "Bern")
print(capital)
```

In [ ]:

```
capital = capitals.pop("France", "Paris")
print(capital)
```

In [ ]:

```
capital = capitals.pop("Germany", "München")
print(capital)
```

popitem() is a method of dict, which doesn't take any parameter and removes and returns an arbitrary (key,value) pair as a 2-tuple. If popitem() is applied on an empty dictionary, a KeyError will be raised.

In [ ]:

```
capitals = {"Springfield": "Illinois",
            "Augusta": "Maine",
            "Boston": "Massachusetts",
            "Lansing": "Michigan",
```

```
              "Albany": "New York",
              "Olympia": "Washington",
              "Toronto": "Ontario"}
(city, state) = capitals.popitem()
(city, state)
```

In [ ]:

```
print(capitals.popitem())
```

In [ ]:

```
print(capitals.popitem())
```

In [ ]:

```
print(capitals.popitem())
```

In [ ]:

```
print(capitals.popitem())
```

ACCESSING NON-EXISTING KEYS

If you try to access a key which doesn't exist, you will get an error message:

In [ ]:

```
locations = {"Toronto": "Ontario", "Vancouver": "British Columbi
a"}
locations["Ottawa"]
```

You can prevent this by using the "in" operator:

In [ ]:

```
province = "Ottawa"

if province in locations:
    print(locations[province])
else:
    print(province + " is not in locations")
```

Another method to access the values via the key consists in using the get() method. get() is not raising an error, if an index doesn't exist. In this case it will return None. It's also possible to set a default value, which will be returned, if an index doesn't exist:

In [ ]:

```
proj_language = {"proj1":"Python", "proj2":"Perl", "proj3":"Java"}
proj_language["proj1"]
```

In [ ]:

```
proj_language.get("proj2")
```

```
proj_language.get("proj4")
print(proj_language.get("proj4"))
```

In [ ]:

```
# setting a default value:
proj_language.get("proj4", "Python")
```

IMPORTANT METHODS

A dictionary can be copied with the method copy():

In [ ]:

```
words = {'house': 'Haus', 'cat': 'Katze'}
w = words.copy()
words["cat"]="chat"
print(w)
```

In [ ]:

```
print(words)
```

This copy is a shallow copy, not a deep copy. If a value is a complex data type like a list, for example, in-place changes in this object have effects on the copy as well:

In [ ]:

```
trainings = { "course1":{"title":"Python Training Course for Begin
ners",
                     "location":"Frankfurt",
                     "trainer":"Steve G. Snake"},
            "course2":{"title":"Intermediate Python Training",
                     "location":"Berlin",
                     "trainer":"Ella M. Charming"},
            "course3":{"title":"Python Text Processing Course",
                     "location":"München",
                     "trainer":"Monica A. Snowdon"}
            }

trainings2 = trainings.copy()

trainings["course2"]["title"] = "Perl Training Course for Beginner
s"
```

```
print(trainings2)
```

If we check the output, we can see that the title of course2 has been changed not only in the dictionary training but in trainings2 as well.

Everything works the way you expect it, if you assign a new value, i.e. a new object, to a key:

In [ ]:
```
trainings = { "course1": {"title": "Python Training Course for Beg
inners",
                          "location": "Frankfurt",
                          "trainer": "Steve G. Snake"},
             "course2": {"title": "Intermediate Python Training",
                          "location": "Berlin",
                          "trainer": "Ella M. Charming"},
             "course3": {"title": "Python Text Processing Cours
e",
                          "location": "München",
                          "trainer": "Monica A. Snowdon"}
             }

trainings2 = trainings.copy()

trainings["course2"] = {"title": "Perl Seminar for Beginners",
                        "location": "Ulm",
                        "trainer": "James D. Morgan"}
print(trainings2["course2"])
```

If you want to understand the reason for this behaviour, we recommend our chapter "Shallow and Deep Copy".

The content of a dictionary can be cleared with the method clear(). The dictionary is not deleted, but set to an empty dictionary:

In [ ]:
```
w.clear()
print(w)
```

UPDATE: MERGING DICTIONARIES

What about concatenating dictionaries, like we did with lists? There is someting similar for dictionaries: the update method update() merges the keys and values of one dictionary into another, overwriting values of the same key:

In [ ]:
```
knowledge = {"Frank": {"Perl"}, "Monica":{"C","C++"}}
knowledge2 = {"Guido":{"Python"}, "Frank":{"Perl", "Python"}}
knowledge.update(knowledge2)
knowledge
```

ITERATING OVER A DICTIONARY

No method is needed to iterate over the keys of a dictionary:

In [ ]:
```
d = {"a":123, "b":34, "c":304, "d":99}
for key in d:
    print(key)
```

However, it's possible to use the method keys(), we will get the same result:

In [ ]:
```
for key in d.keys():
    print(key)
```

The method values() is a convenient way for iterating directly over the values:

In [ ]:
```
for value in d.values():
    print(value)
```

The above loop is logically equivalent to the following one:

In [ ]:
```
for key in d:
    print(d[key])
```

We said logically, because the second way is less efficient!

If you are familiar with the timeit possibility of ipython, you can measure the time used for the two alternatives:

```
%%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for key in d.keys():
    x = d[key]
```

In [ ]:

```
%%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for value in d.values():
    x = value
```

CONNECTION BETWEEN LISTS AND DICTIONARIES

If you have worked for a while with Python, nearly inevitably the moment will come, when you want or have to convert lists into dictionaries or vice versa. It wouldn't be too hard to write a function doing this. But Python wouldn't be Python, if it didn't provide such functionalities.

If we have a dictionary

In [ ]:

```
D = {"list": "Liste", "dictionary": "Wörterbu
ch", "function": "Funktion"}
```

we could turn this into a list with two-tuples:

In [ ]:

```
L = [("list", "Liste"), ("dictionary", "Wörterbuch"), ("functio
n", "Funktion")]
```

The list L and the dictionary D contain the same content, i.e. the information content, or to express "The entropy of L and D is the same" sententiously. Of course, the information is harder to retrieve from the list L than from the dictionary D. To find a certain key in L, we would have to browse through the tuples of the list and compare the first components of the tuples with the key we are looking for. The dictionary search is implicitly implemented for maximum efficiency

LISTS FROM DICTIONARIES

It's possible to create lists from dictionaries by using the methods items(), keys() and values(). As the name implies the method keys() creates a list, which consists solely of the keys of the dictionary. values() produces a list consisting of the values. items() can be used to create a list consisting of 2-tuples of (key,value)-pairs:

In [ ]:

```
w = {"house": "Haus", "cat": "", "red": "rot"}
items_view = w.items()
items = list(items_view)
```

```
items
```

In [ ]:
```
keys_view = w.keys()
keys = list(keys_view)
keys
```

In [ ]:
```
values_view = w.values()
values = list(values_view)
values
```

In [ ]:
```
values_view
```

In [ ]:
```
items_view
```

In [ ]:
```
keys_view
```

If we apply the method items() to a dictionary, we don't get a list back, as it used to be the case in Python 2, but a so-called items view. The items view can be turned into a list by applying the list function. We have no information loss by turning a dictionary into an item view or an items list, i.e. it is possible to recreate the original dictionary from the view created by items(). Even though this list of 2-tuples has the same entropy, i.e. the information content is the same, the efficiency of both approaches is completely different. The dictionary data type provides highly efficient methods to access, delete and change the elements of the dictionary, while in the case of lists these functions have to be implemented by the programmer.

TURN LISTS INTO DICTIONARIES

Now we will turn our attention to the art of cooking, but don't be afraid, this remains a python course and not a cooking course. We want to show you, how to turn lists into dictionaries, if these lists satisfy certain conditions. We have two lists, one containing the dishes and the other one the corresponding countries:

Now we will create a dictionary, which assigns a dish, a country-specific dish, to a country; please forgive us for resorting to the common prejudices. For this purpose, we need the function zip(). The name zip was well chosen, because the two lists get combined, functioning like a zipper. The result is a list iterator.[1] This means that we have to wrap a list() casting function around the zip call to get a list so that we can see what is going on:

In [ ]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_iterator = zip(countries, dishes)
```

```
country_specialities_iterator
```

In [ ]:
```
country_specialities = list(country_specialities_iterator)
print(country_specialities)
```

Alternatively, you could have iteras over the zip object in a for loop. This way we are not creating a list, which is more efficient, if we only want to iterate over the values and don't need a list.

In [ ]:
```
for country, dish in zip(countries, dishes):
    print(country, dish)
```

Now our country-specific dishes are in a list form, - i.e. a list of two-tuples, where the first components are seen as keys and the second components as values - which can be automatically turned into a dictionary by casting it with dict().

In [ ]:
```
country_specialities_dict = dict(country_specialities)
print(country_specialities_dict)
```

Yet, this is very inefficient, because we created a list of 2-tuples to turn this list into a dict. This can be done directly by applying dict to zip:

In [ ]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
dict(zip(countries, dishes))
```

There is still one question concerning the function zip(). What happens, if one of the two argument lists contains more elements than the other one?

It's easy to answer: The superfluous elements, which cannot be paired, will be ignored:

In [ ]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"," Switzerland"]
country_specialities = list(zip(countries, dishes))
country_specialities_dict = dict(country_specialities)
print(country_specialities_dict)
```

So in this course, we will not answer the question about what the national dish of Switzerland is.

EVERYTHING IN ONE STEP

Normally, we recommend not implementing too many steps in one programming expression, though it looks more impressive and the code is more compact. Using "talking" variable names in intermediate steps can enhance legibility. Though it might be alluring to create our previous dictionary just in one go:

In [ ]:
```
country_specialities_dict = dict(zip(["pizza", "sauerkraut", "paella", "hamburger"],
                                     ["Italy", "Germany", "Spain", "USA"," Switzerland"]))
print(country_specialities_dict)
```

Of course, it is more readable like this:

In [ ]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
country_specialities_dict = dict(country_specialities_zip)
print(country_specialities_dict)
```

We get the same result, as if we would have called it in one go.

Especialy for those migrating from Python 2.x to Python 3.x: zip() used to return a list, now it's returning an iterator. You have to keep in mind that iterators exhaust themselves, if they are used. You can see this in the following interactive session:

In [ ]:
```
l1 = ["a","b","c"]
l2 = [1,2,3]
c = zip(l1, l2)
for i in c:
    print(i)
```

This effect can be seen by calling the list casting operator as well:

In [ ]:
```
l1 = ["a","b","c"]
l2 = [1,2,3]
```

```
c = zip(l1,l2)
z1 = list(c)
z2 = list(c)
print(z1)
```

In [ ]:
```
print(z2)
```

As an exercise, you may muse about the following script. Why do we get an empty directory by calling `dict(country_specialities_zip)` ?

In [ ]:
```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
print(list(country_specialities_zip))

country_specialities_dict = dict(country_specialities_zip)
print(country_specialities_dict)
```

EXERCISES

Write a function dict_merge_sum that takes two dictionaries `d1` and `d2` as parameters. The values of both dictionaries are numerical. The function should return the merged sum dictionary `m` of those dictionaries. If a key `k` is both in `d1` and `d2`, the corresponding values will be added and included in the dictionary `m` If `k` is only contained in one of the dictionaries, the `k` and the corresponding value will be included in `m`

Given is the following simplified data of a supermarket:

In [ ]:
```
supermarket = { "milk": {"quantity": 20, "price": 1.19},
                "biscuits":  {"quantity": 32, "price": 1.45},
                "butter":  {"quantity": 20, "price": 2.29},
                "cheese":  {"quantity": 15, "price": 1.90},
                "bread":  {"quantity": 15, "price": 2.59},
                "cookies":  {"quantity": 20, "price": 4.99},
                "yogurt": {"quantity": 18, "price": 3.65},
                "apples":  {"quantity": 35, "price": 3.15},
                "oranges":  {"quantity": 40, "price": 0.99},
```
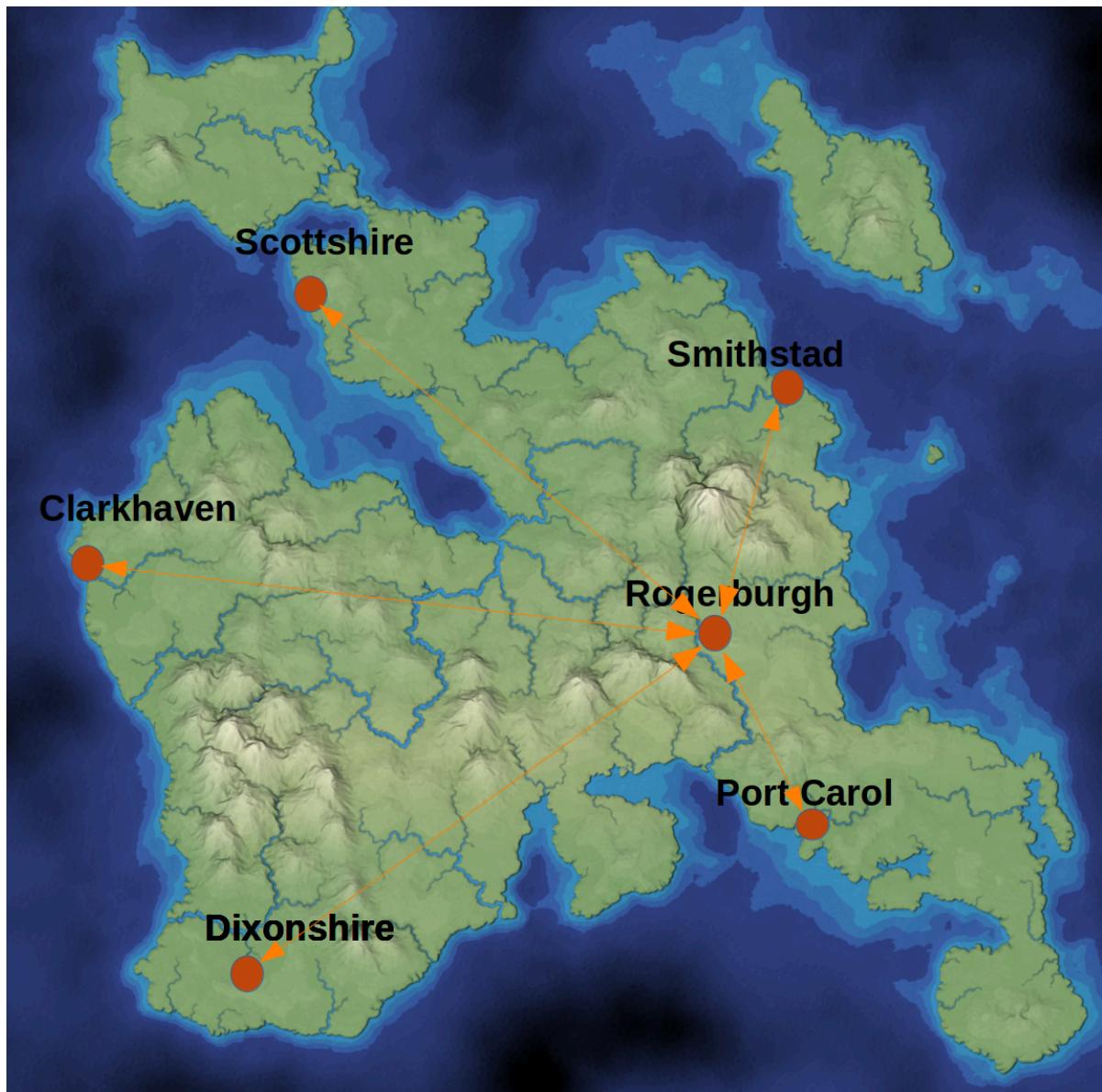
```
                            "bananas": {"quantity": 23, "price": 1.29}}
```

To be ready for an imminent crisis you decide to buy everything. This isn't particularly social behavior, but for the sake of the task, let's imagine it. The question is how much will you have to pay?

<div style="background-color:#EB6C0B; height:40px;"></div>

- Create a virtual supermarket. For every article there is a price per article and a quantity, i.e. the stock. (Hint: you can use the one from the previous exercise!)
- Create shopping lists for customers. The shopping lists contain articles plus the quantity.
- The customers fill their carts, one after the other. Check if enough goods are available! Create a receipt for each customer.

<div style="background-color:#EB6C0B; height:40px;"></div>

Given is the island of Vannoth

Create a

dictionary, where we get for every city of Vannoth the distance to the capital city of Rogeburgh

Create a dictionary where you can get the distance between two arbitrary cities

SOLUTIONS

We offer two solutions to this exercise:

The first one is only using things you will have learned, if you followed our Python course sequentially. The second solution uses techniques which will be covered later in our tutorial.

First solution:

In [ ]:

```python
def dict_merge_sum(d1, d2):
    """ Merging and calculating the sum of two dictionaries:
    Two dicionaries d1 and d2 with numerical values and
    possibly disjoint keys are merged and the values are added if
    the exist in both values, otherwise the missing value is take
n to
    be 0"""

    merged_sum = d1.copy()
    for key, value in d2.items():
        if key in d1:
            d1[key] += value
        else:
            d1[key] = value

    return merged_sum

d1 = dict(a=4, b=5, d=8)
d2 = dict(a=1, d=10, e=9)

dict_merge_sum(d1, d2)
```

Second solution:

In [ ]:

```python
def dict_sum(d1, d2):
    """  Merging and calculating the sum of two dictionaries:
    Two dicionaries d1 and d2 with numerical values and
    possibly disjoint keys are merged and the values are added if
    the exist in both values, otherwise the missing value is take
n to
    be 0"""
```

```
    return { k: d1.get(k, 0) + d2.get(k, 0) for k in set(d1) | se
t(d2) }

d1 = dict(a=4, b=5, d=8)
d2 = dict(a=1, d=10, e=9)

dict_merge_sum(d1, d2)
```

In [ ]:
```
total_value = 0
for article in supermarket:
    quantity = supermarket[article]["quantity"]
    price = supermarket[article]["price"]
    total_value += quantity * price

print(f"The total price for buying everything: {total_value:7.2
f}")
```

In [ ]:
```
supermarket = { "milk": {"quantity": 20, "price": 1.19},
                "biscuits":  {"quantity": 32, "price": 1.45},
                "butter":   {"quantity": 20, "price": 2.29},
                "cheese":   {"quantity": 15, "price": 1.90},
                "bread":   {"quantity": 15, "price": 2.59},
                "cookies":  {"quantity": 20, "price": 4.99},
                "yogurt": {"quantity": 18, "price": 3.65},
                "apples":  {"quantity": 35, "price": 3.15},
                "oranges":  {"quantity": 40, "price": 0.99},
                "bananas": {"quantity": 23, "price": 1.2
9}
              }

customers = ["Frank", "Mary", "Paul"]
shopping_lists = {
   "Frank" : [('milk', 5), ('apples', 5), ('butter', 1), ('cookie
s', 1)],
   "Mary":  [('apples', 2), ('cheese', 4), ('bread', 2), ('pear
s', 3),
```

```
                   ('bananas', 4), ('oranges', 1), ('cherries', 4)],
   "Paul":  [('biscuits', 2), ('apples', 3), ('yogurt', 2), ('pear
s', 1),
                   ('butter', 3), ('cheese', 1), ('milk', 1), ('cookie
s', 4)]}


# filling the carts
carts = {}
for customer in customers:
    carts[customer] = []
    for article, quantity in shopping_lists[customer]:
        if article in supermarket:
            if supermarket[article]["quantity"] < quantity:
                quantity = supermarket[article]["quantity"]
            if quantity:
                supermarket[article]["quantity"] -= quantity
                carts[customer].append((article, quantity))
for customer in customers:
     print(carts[customer])


print("checkout")
for customer in customers:
    print("\ncheckout for " + customer + ":")
    total_sum = 0
    for name, quantity in carts[customer]:
        unit_price = supermarket[name]["price"]
        item_sum = quantity * unit_price
        print(f"{quantity:3d} {name:12s} {unit_price:8.2f} {item_s
um:8.2f}")
        total_sum += item_sum
    print(f"Total sum:              {total_sum:11.2f}")
```

Alternative solution to exercise 3, in which we create the chopping lists randomly:

In [ ]:

```
import random

supermarket = { "milk": {"quantity": 20, "price": 1.19},
                "biscuits":  {"quantity": 32, "price": 1.45},
                "butter":  {"quantity": 20, "price": 2.29},
                "cheese":  {"quantity": 15, "price": 1.90},
                "bread":  {"quantity": 15, "price": 2.59},
```

```python
                "cookies":  {"quantity": 20, "price": 4.99},
                "yogurt": {"quantity": 18, "price": 3.65},
                "apples":  {"quantity": 35, "price": 3.15},
                "oranges":  {"quantity": 40, "price": 0.99},
                "bananas": {"quantity": 23, "price": 1.2
9}
            }


articles4shopping_lists = list(supermarket.keys()) + ["pears", "ch
erries"]

max_articles_per_customer = 5 # not like a real supermarket :-)
customers = ["Frank", "Mary", "Paul", "Jennifer"]
shopping_lists = {}
for customer in customers:
    no_of_items = random.randint(1, len(articles4shopping_lists))
    shopping_lists[customer] = []
    for article_name in random.sample(articles4shopping_lists, n
o_of_items):
        quantity = random.randint(1, max_articles_per_customer)
        shopping_lists[customer].append((article_name, quantity))

# let's look at the shopping lists
print("Shopping lists:")
for customer in customers:
    print(customer + ":  " + str(shopping_lists[customer]))

# filling the carts
carts = {}
for customer in customers:
    carts[customer] = []
    for article, quantity in shopping_lists[customer]:
        if article in supermarket:
            if supermarket[article]["quantity"] < quantity:
                quantity = supermarket[article]["quantity"]
            if quantity:
                supermarket[article]["quantity"] -= quantity
                carts[customer].append((article, quantity))


print("\nCheckout")
for customer in customers:
    print("\ncheckout for " + customer + ":")
    total_sum = 0
```

```
    for name, quantity in carts[customer]:
        unit_price = supermarket[name]["price"]
        item_sum = quantity * unit_price
        print(f"{quantity:3d} {name:12s} {unit_price:8.2f} {item_s
um:8.2f}")
        total_sum += item_sum
    print(f"Total sum:              {total_sum:11.2f}")
```

In [ ]:

```
rogerburgh = {'Smithstad': 5.2, 'Scottshire': 12.3, 'Clarkhaven':
14.9, 'Dixonshire': 12.7, 'Port Carol': 3.4}
```

In [ ]:

```
distances = {'Rogerburgh': {'Smithstad': 5.2, 'Scottshire': 12.3,
'Clarkhaven': 14.9, 'Dixonshire': 12.7, 'Port Carol': 3.4},
             'Smithstad': {'Rogerburh': 5.2, 'Scottshire': 11.1,
'Clarkhaven': 19.1, 'Dixonshire': 17.9, 'Port Carol': 8.6},
             'Scottshire': {'Smithstad': 11.1, 'Rogerburgh': 1
2.3, 'Clarkhaven': 18.1, 'Dixonshire': 19.3, 'Port Carol': 15.7},
             'Clarkshaven': {'Smithstad': 19.1, 'Scottshire': 1
8.1, 'Rogerburgh': 14.9, 'Dixonshire': 6.8, 'Port Carol': 17.1},
             'Dixonshire': {'Smithstad': 5.2, 'Scottshire': 12.3,
'Clarkhaven': 14.9, 'Rogerburg': 12.7, 'Port Carol': 16.1},
             'Port Carol': {'Smithstad': 8.6, 'Scottshire': 15.7,
'Clarkhaven': 17.1, 'Dixonshire': 16.1, 'Rogerburgh': 3.4}
             }
```

FOOTNOTES

[1] If you would like to learn more about how iterators work and how to use them, we recommend that you go through our Iterators and Generators chapter.

INTRODUCTION

In this chapter of our Python tutorial we deal with the wonderful and extremely useful functionality 'zip'. Unfortunately, beginners of Python are quite often unnecessarily confused and frightened by zip. First of all the name is confusing. Lots of people confuse the zip of Python with the well-known archive file format ZIP, which is used for lossless data compression. Since this data compression not only keeps the required storage space very small, but is also carried out very quickly, the name 'zip' was chosen. In this context, "zip" means "to act quickly or move quickly".

The 'zip' of Python has absolutely nothing to do with it. The functionality 'zip' in Python is based on another meaning of the English word 'zip', "closing something with a zipper".

We will see that zip can be very easy to understand. However, this does not apply if you come up with the idea of using the Python help function. The help text is certainly partly responsible for this and frightens the novices away:

```
Return a zip object whose .next() me
thod returns a tuple where the i-th
element comes from the i-th iterabl
e argument. The .next() method conti
nues until the shortest iterable in
the argument sequence is exhausted a
nd then it raises StopIteration.
```
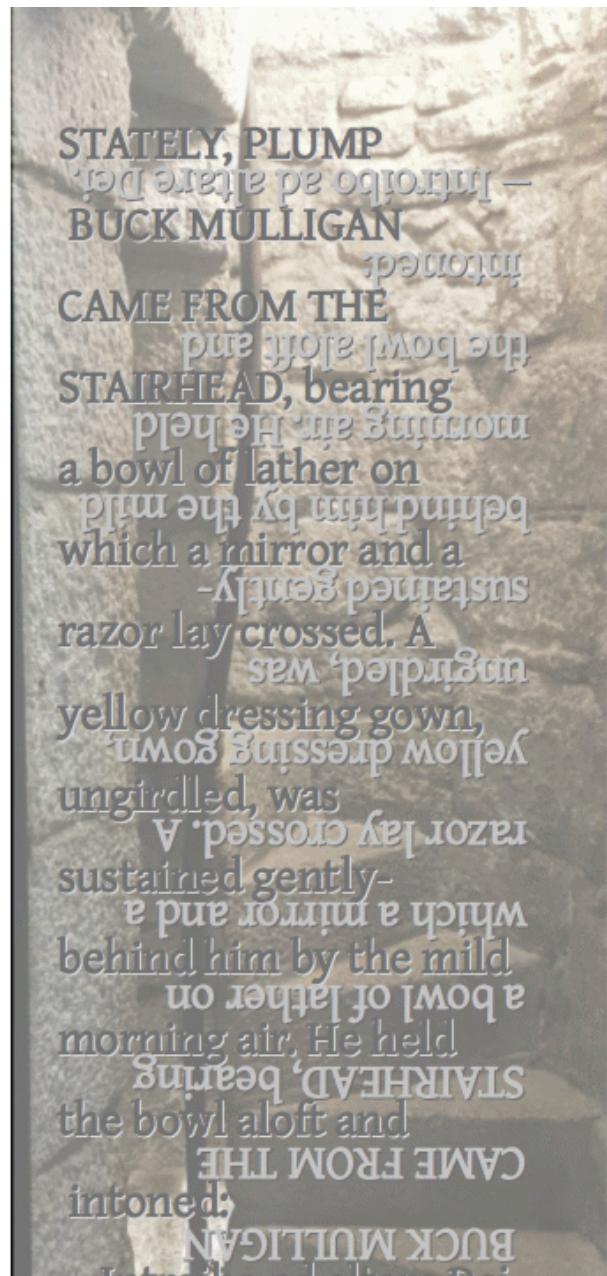
This text is correct but it is hard to understand. Let us start with simple examples before we try to simplify the theory behind zip.

We will start with two lists and apply zip to those two lists:

```python
a_couple_of_letters = ["a", "b",
"c", "d", "e", "f"]
some_numbers = [5, 3, 7, 9, 11, 2]

print(zip(a_couple_of_letters, som
e_numbers))
```

```
<zip object at 0x7efc8724f9b0>
```

The result is a zip object and you are not wiser than before, if you don't know zip. The application of zip returns an iterator, which is capable of producing tuples. It is combining the first items of each iterable (in our example lists) into a tuple, after this it combines the second items and so on. It stops when one of them is exhausted, i.e. there are no more items available.
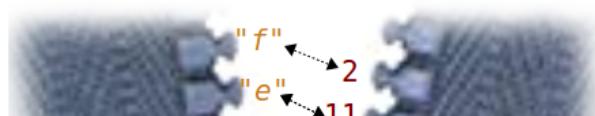
The best way to see what it creates is to use it in a for loop.

```python
for t in zip(a_couple_of_letters, some_numbers):
    print(t)
```

```
('a', 5)
('b', 3)
('c', 7)
('d', 9)
('e', 11)
('f', 2)
```

If you look closely at the output above and the following picture, you will hopefully understand why zip has something to do with a zipper and why they had chosen the name.

zip can have an arbitrary number of iterable arguments as we can see in the following example:


```python
location = ["Helgoland", "Kiel",
           "Berlin-Tegel", "Konstanz",
           "Hohenpeißenberg"]
air_pressure = [1021.2, 1019.9, 1023.7, 1023.1, 1027.7]
temperatures = [6.0, 4.3, 2.7, -1.4, -4.4]
altitude = [4, 27, 37, 443, 977]

for t in zip(location, air_pressure, temperatures, altitude):
    print(t)
```

```
('Helgoland', 1021.2, 6.0, 4)
('Kiel', 1019.9, 4.3, 27)
('Berlin-Tegel', 1023.7, 2.7, 37)
('Konstanz', 1023.1, -1.4, 443)
('Hohenpeißenberg', 1027.7, -4.4, 977)
```

The use of zip is not restricted to lists and tuples. It can be applied to all iterable objects like lists, tuples,

strings, dictionaries, sets, range and many more of course.

```
food = ["ham", "spam", "cheese"]

for item in zip(range(1000, 1003), food):
    print(item)
```

```
(1000, 'ham')
(1001, 'spam')
(1002, 'cheese')
```

CALLING ZIP WITH NO ARGUMENT

Let us have a look at what happens, if we call zip without any parameter:

```
for i in zip():
    print("This will not be printed")
```

The body of the loop was not being executed!

CALLING ZIP WITH ONE ITERABLE

What happens if we call zip with just one argument, e.g. a string:

```
s = "Python"
for t in zip(s):
    print(t)
```

```
('P',)
('y',)
('t',)
('h',)
('o',)
('n',)
```

So this call creates an iterator which produces tuples with one single element, in our case the characters of the string.

PARAMETERS WITH DIFFERENT LENGTHS

As we have seen, zip can be called with an arbitrary number of iterable objects as arguments. So far the number of elements or the length of these iterables had been the same. This is not necessary. If the Lengths are different, zip will stop producing an output as soon as one of the argument sequences is exhausted. "stop

producing an output" actually means it will raise a StopIteration exception like all other iterators do. This exception is caught by the for loop. You will find more about this way of working in the chapter "Iterators and Iterables" of our Python tutorial.

The following example is a zip call with two list with different lenghts:

```python
colors = ["green", "red", "blue"]
cars = ["BMW", "Alfa Romeo"]

for car, color in zip(cars, colors):
    print(car, color)
```

```
BMW green
Alfa Romeo red
```

ADVANCED USAGES OF ZIP

We have a list with the six largest cities in Switzerland. It consists of tuples with the pairs, city and population number:

```python
cities_and_population = [("Zurich", 415367),
                         ("Geneva", 201818),
                         ("Basel", 177654),
                         ("Lausanne", 139111),
                         ("Bern", 133883),
                         ("Winterthur", 111851)]
```

The task consists of creating two lists: One with the city names and one with the population numbers. zip is the solution to this problem, but we also have to use the star operator to unpack the list:

```python
cities, populations = list(zip(*cities_and_population))
print(cities)
print(populations)
```

```
('Zurich', 'Geneva', 'Basel', 'Lausanne', 'Bern', 'Winterthur')
(415367, 201818, 177654, 139111, 133883, 111851)
```

This is needed e.g., if we want to plot these data. The following example is just used for illustrating purposes. It is not necessary to understand it completely. You have to be familiar with Pandas. In the following program, we assume that only the combined list is available at the beginning:

```python
import pandas as pd

cities_and_population = [("Zurich", 415367),
```
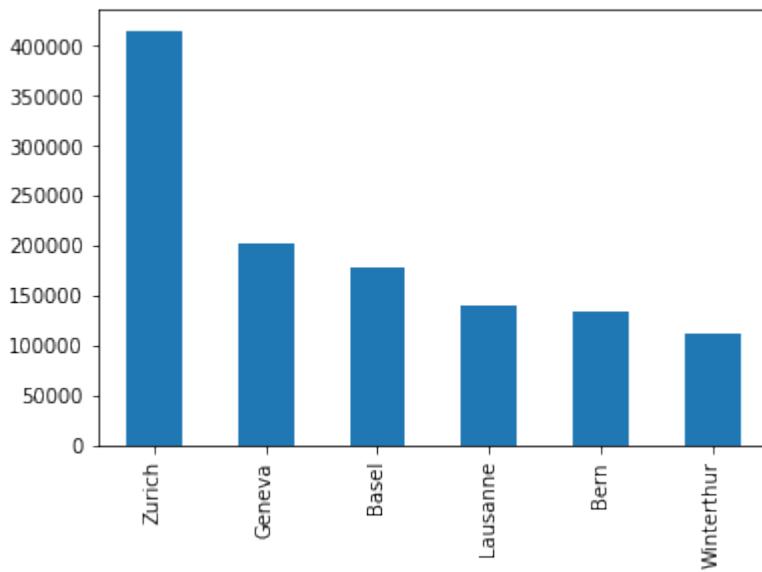
```
                              ("Geneva", 201818),
                              ("Basel", 177654),
                              ("Lausanne", 139111),
                              ("Bern", 133883),
                              ("Winterthur", 111851)]

cities, populations = list(zip(*cities_and_population))
s = pd.Series(populations, index=cities)

s.plot(kind="bar")
```

Output: `<matplotlib.axes._subplots.AxesSubplot at 0x7fdd6494e490>`

CONVERTING TWO ITERABLES INTO A DICTIONARY

zip also offers us an excellent opportunity to convert two iterables into a dictionary. Of course, only if these iterables meet certain requirements. The iterable, which should be used as keys, must be unique and can only consist of immutables. We demonstrate this with the following morse code example. The full morse code alphabet can be found in our chapter on dictionaries:

```
abc = "abcdef"
morse_chars = [".-", "-...", "-.-.", "-..", ".", "..-."]

text2morse = dict(zip(abc, morse_chars))
print(text2morse)

{'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.', 'f':
'..-.'}
```

FOOTNOTES

[1] The data are from the 1st of April 2020, no joke. The locations are chosen arbitrarily. Yet, there is a special reason, why I had chosen Hohenpeißenberg. On the Hohenpeißenberg I was invited to give two unforgettable Python courses for the German weather service: Great people and a fantastic view!

SETS AND FROZENSETS
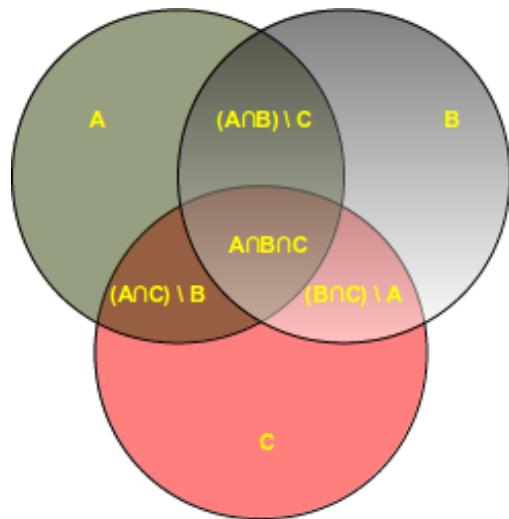
INTRODUCTION

In this chapter of our tutorial, we are dealing with Python's implementation of sets. Though sets are nowadays an integral part of modern mathematics, this has not always been the case. The set theory had been rejected by many, even by some great thinkers. One of them was the philosopher Wittgenstein. He didn't like the set theory and complained mathematics is "ridden through and through with the pernicious idioms of set theory...". He dismissed the set theory as "utter nonsense", as being "laughable" and "wrong". His criticism appeared years after the death of the German mathematician Georg Cantor, the founder of the set theory. David Hilbert defended it from its critics by famously declaring: "No one shall expel us from the Paradise that Cantor has created.

Cantor defined a set at the beginning of his "Beiträge zur Begründung der transfiniten Mengenlehre" as: "A set is a gathering together into a whole of definite, distinct objects of our perception and of our thought - which are called elements of the set." Nowadays, we can say in "plain" English: A set is a well-defined collection of objects.

The elements or members of a set can be anything: numbers, characters, words, names, letters of the alphabet, even other sets, and so on. Sets are usually denoted with capital letters. This is not the exact mathematical definition, but it is good enough for the following.

The data type "set", which is a collection type, has been part of Python since version 2.4. A set contains an unordered collection of unique and immutable objects. The set data type is, as the name implies, a Python implementation of the sets as they are known from mathematics. This explains, why sets unlike lists or tuples can't have multiple occurrences of the same element.

SETS

If we want to create a set, we can call the built-in set function with a sequence or another iterable object.

In the following example, a string is singularized into its characters to build the resulting set x:

```python
x = set("A Python Tutorial")
x
```

Output:
```
{' ', 'A', 'P', 'T', 'a', 'h', 'i', 'l', 'n', 'o', 'r', 't',
'u', 'y'}
```

```python
type(x)
```

set

We can pass a list to the built-in set function, as we can see in the following:

```
x = set(["Perl", "Python", "Java"])
x
```

Output: {'Java', 'Perl', 'Python'}

Now, we want to show what happens, if we pass a tuple with reappearing elements to the set function - in our example the city "Paris":

```
cities = set(("Paris", "Lyon", "London","Berlin","Paris","Birmingh
am"))
cities
```

Output: {'Berlin', 'Birmingham', 'London', 'Lyon', 'Paris'}

As we have expected, no doubles occur in the resulting set of cities.

IMMUTABLE SETS

Sets are implemented in a way, which doesn't allow mutable objects. The following example demonstrates that we cannot include, for example, lists as elements:

```
cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))

cities
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-5a60d6eeb901> in <module>
----> 1 cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))
        2
        3 cities

TypeError: unhashable type: 'list'
```

Tuples on the other hand are fine:

In [ ]:
```
cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))
```

FROZENSETS

Though sets can't contain mutable objects, sets are mutable:

```python
cities = set(["Frankfurt", "Basel","Freiburg"])
cities.add("Strasbourg")
cities
```

Output: `{'Basel', 'Frankfurt', 'Freiburg', 'Strasbourg'}`

Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

```python
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
cities.add("Strasbourg")
```

```
---------------------------------------------------------------------
AttributeError                              Traceback (most recent call last)
<ipython-input-10-9abedff8e0b3> in <module>
      1 cities = frozenset(["Frankfurt", "Basel","Freiburg"])
----> 2 cities.add("Strasbourg")

AttributeError: 'frozenset' object has no attribute 'add'
```

IMPROVED NOTATION

We can define sets (since Python2.6) without using the built-in set function. We can use curly braces instead:

```python
adjectives = {"cheap","expensive","inexpensive","economical"}
adjectives
```

Output: `{'cheap', 'economical', 'expensive', 'inexpensive'}`

SET OPERATIONS

ADD(ELEMENT)

A method which adds an element to a set. This element has to be immutable.

```python
colours = {"red","green"}
colours.add("yellow")
colours
```

Output: `{'green', 'red', 'yellow'}`

```
colours.add(["black","white"])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-b914ecb44006> in <module>
----> 1 colours.add(["black","white"])

TypeError: unhashable type: 'list'
```

Of course, an element will only be added, if it is not already contained in the set. If it is already contained, the method call has no effect.

All elements will be removed from a set.

```
cities = {"Stuttgart", "Konstanz", "Freiburg"}
cities.clear()
cities
```
Output: `set()`

Creates a shallow copy, which is returned.

```
more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
cities_backup = more_cities.copy()
more_cities.clear()
cities_backup
```
Output: `{'Schaffhausen', 'St. Gallen', 'Winterthur'}`

Just in case, you might think, an assignment might be enough:

```
more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
cities_backup = more_cities
more_cities.clear()
cities_backup
```
Output: `set()`

The assignment "cities_backup = more_cities" just creates a pointer, i.e. another name, to the same data

structure.

This method returns the difference of two or more sets as a new set, leaving the original set unchanged.

```python
x = {"a","b","c","d","e"}
y = {"b","c"}
z = {"c","d"}
x.difference(y)
```
Output: {'a', 'd', 'e'}

```python
x.difference(y).difference(z)
```
Output: {'a', 'e'}

Instead of using the method difference, we can use the operator "-":

```python
x - y
```
Output: {'a', 'd', 'e'}

```python
x - y - z
```
Output: {'a', 'e'}

The method difference_update removes all elements of another set from this set. x.difference_update(y) is the same as "x = x - y" or even x -= y works.

```python
x = {"a","b","c","d","e"}
y = {"b","c"}
x.difference_update(y)
x = {"a","b","c","d","e"}
y = {"b","c"}
x = x - y
x
```
Output: {'a', 'd', 'e'}

An element el will be removed from the set, if it is contained in the set. If el is not a member of the set, nothing will be done.

```
x = {"a","b","c","d","e"}
x.discard("a")
x
```

Output: `{'b', 'c', 'd', 'e'}`

```
x.discard("z")
x
```

Output: `{'b', 'c', 'd', 'e'}`

Works like discard(), but if el is not a member of the set, a KeyError will be raised.

```
x = {"a","b","c","d","e"}
x.remove("a")
x
```

Output: `{'b', 'c', 'd', 'e'}`

```
x.remove("z")
```

```
---------------------------------------------------------------------------
KeyError                                      Traceback (most recent call last)
<ipython-input-18-7c787df95113> in <module>
----> 1 x.remove("z")

KeyError: 'z'
```

This method returns the union of two sets as a new set, i.e. all elements that are in either set.

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x.union(y)
```

Output: `{'a', 'b', 'c', 'd', 'e', 'f', 'g'}`

This can be abbreviated with the pipe operator "|":

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x | y
```
Output: `{'a', 'b', 'c', 'd', 'e', 'f', 'g'}`

Returns the intersection of the instance set and the set s as a new set. In other words, a set with all the elements which are contained in both sets is returned.

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x.intersection(y)
```
Output: `{'c', 'd', 'e'}`

This can be abbreviated with the ampersand operator "&":

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x  & y
```
Output: `{'c', 'd', 'e'}`

```
    This method returns True if two sets have a null intersection.
```

```
x = {"a","b","c"}
y = {"c","d","e"}
x.isdisjoint(y)
```
Output: `False`

```
x = {"a","b","c"}
y = {"d","e","f"}
x.isdisjoint(y)
```
Output: `True`

x.issubset(y) returns True, if x is a subset of y. "<=" is an abbre
viation for "Subset of" and ">=" for "superset of"
"<" is used to check if a set is a proper subset of a set.

```python
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issubset(y)
```

Output: False

```python
y.issubset(x)
```

Output: True

```python
x < y
```

Output: False

```python
y < x # y is a proper subset of x
```

Output: True

```python
x < x # a set can never be a proper subset of oneself.
```

Output: False

```python
x <= x
```

Output: True

x.issuperset(y) returns True, if x is a superset of y. ">=" is an a
bbreviation for "issuperset of"
">" is used to check if a set is a proper superset of a set.

```python
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issuperset(y)
```

Output: True

```python
x > y
```

Output: True

```
x >= y
```
Output: `True`

```
x >= x
```
Output: `True`

```
x > x
```
Output: `False`

```
x.issuperset(x)
```
Output: `True`

pop() removes and returns an arbitrary set element. The method raises a KeyError if the set is empty.

```
x = {"a","b","c","d","e"}
x.pop()
```
Output: `'e'`

```
x.pop()
```
Output: `'a'`

AN EXTENSIVE EXAMPLE FOR SETS

PYTHON AND THE BEST NOVEL

This chapter deals with natural languages and literature. It will be also an extensive example and use case for Python sets. Novices in Python often think that sets are just a toy for mathematicians and that there is no real use case in programming. The contrary is true. There are multiple use cases for sets. They are used, for example, to get rid of doublets - multiple occurrences of elements - in a list, i.e. to make a list unique.

In the following example we will use sets to determine the different words occurring in a novel. Our use case is build around a novel which has been praised by many, and regarded as the best novel in the English language and also as the hardest to read. We are talking about the novel "Ulysses" by James Joyce. We will not talk about or examine the beauty of the language or the language style. We will study the novel by having a close look at the words used in the novel. Our approach will be purely statitical. The claim is that James Joyce used in his novel more words than any other author. Actually his vocabulary is above and beyond all other authors, maybe even Shakespeare.

Besides Ulysses we will use the novels "Sons and Lovers" by D.H. Lawrence, "The Way of All Flesh" by Samuel Butler, "Robinson Crusoe" by Daniel Defoe, "To the Lighthouse" by Virginia Woolf, "Moby Dick" by Herman Melville and the Short Story "Metamorphosis" by Franz Kafka.

Before you continue with this chapter of our tutorial it might be a good idea to read the chapter Sets and Frozen Sets and the two chapter on regular expressions and advanced regular expressions.

DIFFERENT WORDS OF A TEXT

To cut out all the words of the novel "Ulysses" we can use the function findall from the module "re":

```python
import re

# we don't care about case sensitivity and therefore use lower:
ulysses_txt = open("books/james_joyce_ulysses.txt").read().lower()

words = re.findall(r"\b[\w-]+\b", ulysses_txt)
print("The novel ulysses contains " + str(len(words)))
```

```
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-1-18646764c8d9> in <module>
      2
      3 # we don't care about case sensitivity and therefore use lower:
----> 4 ulysses_txt = open("books/james_joyce_ulysses.txt").read().lower()
      5
      6 words = re.findall(r"\b[\w-]+\b", ulysses_txt)

~\Anaconda3\lib\encodings\cp1252.py in decode(self, input, final)
     21 class IncrementalDecoder(codecs.IncrementalDecoder):
     22     def decode(self, input, final=False):
---> 23         return codecs.charmap_decode(input,self.errors,decoding_tab
le)[0]
     24
     25 class StreamWriter(Codec,codecs.StreamWriter):

UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in position 2706
53: character maps to <undefined>
```

This number is the sum of all the words, together with the many words that occur multiple times:

```python
for word in ["the", "while", "good", "bad", "ireland", "irish"]:
    print("The word '" + word + "' occurs " + \
          str(words.count(word)) + " times in the novel!" )
```

```
The word 'the' occurs 15112 times in the novel!
The word 'while' occurs 123 times in the novel!
The word 'good' occurs 321 times in the novel!
The word 'bad' occurs 90 times in the novel!
The word 'ireland' occurs 90 times in the novel!
The word 'irish' occurs 117 times in the novel!
```

272452 surely is a huge number of words for a novel, but on the other hand there are lots of novels with even more words. More interesting and saying more about the quality of a novel is the number of different words. This is the moment where we will finally need "set". We will turn the list of words "words" into a set. Applying "len" to this set will give us the number of different words:

```python
diff_words = set(words)
print("'Ulysses' contains " + str(len(diff_words)) + " different w
ords!")
```

```
'Ulysses' contains 29422 different words!
```

This is indeed an impressive number. You can see this, if you look at the other novels below:

```python
novels = ['sons_and_lovers_lawrence.txt',
          'metamorphosis_kafka.txt',
          'the_way_of_all_flash_butler.txt',
          'robinson_crusoe_defoe.txt',
          'to_the_lighthouse_woolf.txt',
          'james_joyce_ulysses.txt',
          'moby_dick_melville.txt']

for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    diff_words = set(words)
    n = len(diff_words)
    print("{name:38s}: {n:5d}".format(name=novel[:-4], n=n))
```

```
sons_and_lovers_lawrence               : 10822
metamorphosis_kafka                    :  3027
the_way_of_all_flash_butler            : 11434
robinson_crusoe_defoe                  :  6595
to_the_lighthouse_woolf                : 11415
james_joyce_ulysses                    : 29422
moby_dick_melville                     : 18922
```

SPECIAL WORDS IN ULYSSES

We will subtract all the words occurring in the other novels from "Ulysses" in the following little Python program. It is amazing how many words are used by James Joyce and by none of the other authors:

```python
words_in_novel = {}
for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words

words_only_in_ulysses = set(words_in_novel['james_joyce_ulysses.t
xt'])
novels.remove('james_joyce_ulysses.txt')
for novel in novels:
    words_only_in_ulysses -= set(words_in_novel[novel])

with open("books/words_only_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_only_in_ulysses)
```

```
    fh.write(txt)

print(len(words_only_in_ulysses))
```

```
 15314
```

By the way, Dr. Seuss wrote a book with only 50 different words: "Green Eggs and Ham"

The file with the words only occurring in Ulysses contains strange or seldom used words like:

huntingcrop tramtrack pappin kithogue pennyweight undergarments scission nagyaságos wheedling begad dogwhip hawthornden turnbull calumet covey repudiated pendennis waistcoatpocket nostrum

COMMON WORDS

It is also possible to find the words which occur in every book. To accomplish this, we need the set intersection:

```
# we start with the words in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

print(len(common_words))
```

```
 1745
```

DOING IT RIGHT

We made a slight mistake in the previous calculations. If you look at the texts, you will notice that they have a header and footer part added by Project Gutenberg, which doesn't belong to the texts. The texts are positioned between the lines:

START OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

and

END OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

or

START OF THIS PROJECT GUTENBERG EBOOK ULYSSES

and

END OF THIS PROJECT GUTENBERG EBOOK ULYSSES

The function read_text takes care of this:

```python
def read_text(fname):
    beg_e = re.compile(r"\*\*\* ?start of (this|the) project guten
berg ebook[^*]*\*\*\*")
    end_e = re.compile(r"\*\*\* ?end of (this|the) project gutenbe
rg ebook[^*]*\*\*\*")
    txt = open("books/" + fname).read().lower()
    beg = beg_e.search(txt).end()
    end = end_e.search(txt).start()
    return txt[beg:end]

words_in_novel = {}
for novel in novels + ['james_joyce_ulysses.txt']:
    txt = read_text(novel)
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words

words_in_ulysses =  set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    words_in_ulysses -= set(words_in_novel[novel])

with open("books/words_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_in_ulysses)
    fh.write(txt)

print(len(words_in_ulysses))


# we start with the words in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

print(len(common_words))
```

```
15341
1279
```

The words of the set "common_words" are words belong to the most frequently used words of the English language. Let's have a look at 30 arbitrary words of this set:

```python
counter = 0
for word in common_words:
```

```
    print(word, end=", ")
    counter += 1
    if counter == 30:
        break
```

```
send, such, mentioned, writing, found, speak, fond, food, their, m
other, household, through, prepared, flew, gently, work, station,
naturally, near, empty, filled, move, unknown, left, alarm, listen
ing, waited, showed, broke, laugh,
```

ancient, broke, breathing, laugh, divided, forced, wealth, ring, outside, throw, person, spend, better, errand, school, sought, knock, tell, inner, run, packed, another, since, touched, bearing, repeated, bitter, experienced, often, one,

INPUT FROM KEYBOARD

THE INPUT FUNCTION

There are hardly any programs without any input. Input can come in various ways, for example, from a database, another computer, mouse clicks and movements or from the internet. Yet, in most cases the input stems from the keyboard. For this purpose, Python provides the function input(). input has an optional parameter, which is the prompt string.

If the input function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key. The text of the optional parameter, i.e. the prompt, will be printed on the screen.

The input of the user will be returned as a string without any changes. If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the eval function.

Let's have a look at the following example:

```python
name = input("What's your name? ")
print("Nice to meet you " + name + "!")
```

```
What's your name? Melisa
Nice to meet you Melisa!
```
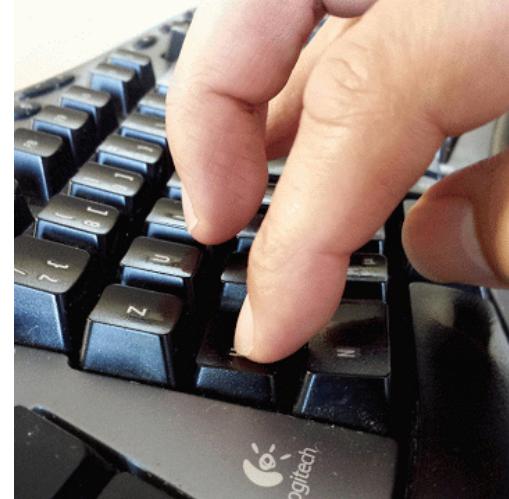
```python
age = input("Your age? ")
print("So, you are already " + age + " years old, " + name + "!")
```

```
Your age? 26
So, you are already 26 years old, Melisa!
```

We save the program as "input_test.py" and run it:

```
 $ python input_test.py
What's your name? "Frank"
Nice to meet you Frank!
Your age? 42
So, you are already 42 years old, Frank!
```

We will further experiment with the input function in the following interactive Python session:

```
cities_canada = input("Largest cities in Canada: ")
print(cities_canada, type(cities_canada))
```

```
Largest cities in Canada: Toronto, Calgary, Montreal, Ottawa
Toronto, Calgary, Montreal, Ottawa <class 'str'>
```

```
cities_canada = eval(input("Largest cities in Canada: "))
print(cities_canada, type(cities_canada))
```

```
Largest cities in Canada: Toronto, Calgary, Montreal, Ottawa
```

```
---------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-4febe85dfcfd> in <module>
----> 1 cities_canada = eval(input("Largest cities in Canada: "))
      2 print(cities_canada, type(cities_canada))

<string> in <module>

NameError: name 'Toronto' is not defined
```

```
population = input("Population of Toronto? ")
print(population, type(population))
```

```
Population of Toronto? 2615069
2615069 <class 'str'>
```
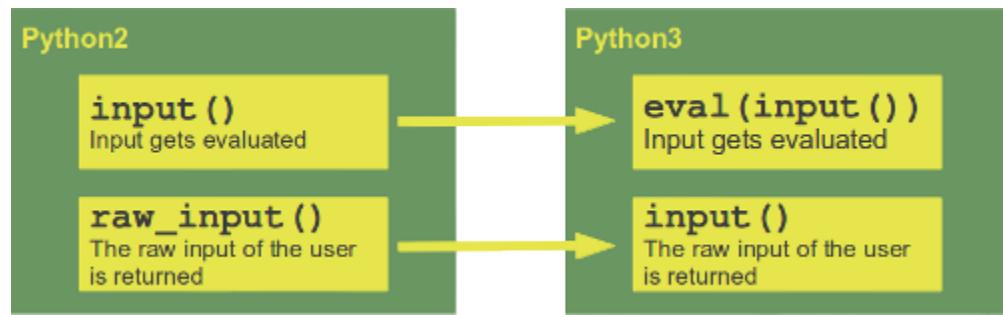
```
population = int(input("Population of Toronto? "))
print(population, type(population))
```

```
Population of Toronto? 2615069
2615069 <class 'int'>
```

DIFFERENCES BETWEEN PYTHON2 AND PYTHON3

The usage of input or better the implicit evaluation of the input has often lead to serious programming mistakes in the earlier Python versions, i.e. 2.x Therefore, in Python 3 the input function behaves like the raw_input function from Python2.

The changes between the versions are illustrated in the following diagram:

CONDITIONAL STATEMENTS
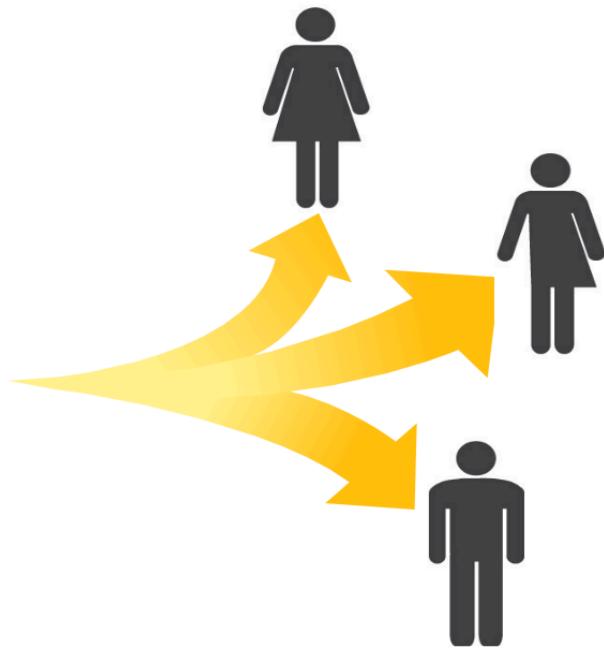
DECISION MAKING

Do you belong to those people who find it hard to make decisions? In real life, I mean. In certain circumstances, some decisions in daily life are unavoidable, e.g. people have to go their separate ways from time to time, as our picture shows. We all have to make decisions all the time, ranging from trivial issues like choosing what to wear in the morning, what to have for breakfast, right up to life-changing decisions like taking or changing a job, deciding what to study, and many more. However, there you are, having decided to learn more about the conditional statements in Python.

Conditions - usually in the form of if statements - are one of the key features of a programming language, and Python is no exception. You will hardly find a programming language without an if statement.[1] There is hardly a way to program without branches in the code flow, at least if the code needs to solve a useful problem.

A decision must be made when the script or program comes to a point where there is a selection of actions, i.e. different calculations from which to choose.

The decision, in most cases, depends on the value of variables or arithmetic expressions. These expressions are evaluated using the Boolean True or False values. The instructions for decision making are called conditional statements. In a programming language, therefore, the parts of the conditional statements are executed under the given conditions. In many cases there are two code parts: One which will be executed, if the condition is True, and another one, if it is False. In other words, a branch determines which of two (or even more) program parts (alternatives) will be executed depending on one (or more) conditions. Conditional statements and branches belong to the control structures of programming languages, because with their help a program can react to different states that result from inputs and calculations.

CONDITIONAL STATEMENTS IN REAL LIFE

The principle of the indentation style is also known in natural languages, as we can deduce from the following text:

```
 If it rains tomorrow, I'll clean up the baseme
nt.
 After that, I will paint the walls. If there i
s
 any time left, I will file my tax return.
```

Of course, as we all know, there will be no time left to file the tax return. Jokes aside: in the previous text, as you see, we have a sequence of actions that must be performed in chronological order. If you take a closer look at the text, you will find some ambiguity in it: is 'the painting of the walls' also related to the event of the rain? Will the tax declaration be done with or without rain? What will this person do, if it does not rain? Therefore, we extend the text with additional alternatives:

```
If it rains tomorrow, I'll clean up the basement.
After that I will paint the walls. If there is
any time left, I will file my tax return. Otherwise,
i.e. if it will not rain, I will go swimming.
In the evening, I will go to the cinema with my wife!
```

It is still nor clear. Can his wife hope to be invited to the cinema? Does she have to pray or hope for rain? To make the text clear, we can phrase it to be closer to the programming code and the Python coding style, and hopefully there will be a happy ending for his wife:

```
If it rains tomorrow, I will do the following:
    - Clean up the basement
    - Paint the walls
    - If there is any time left, I will make my
      tax declaration
Otherwise I will do the following:
    - Go swimming
Going to the movies in the evening with my wife
```
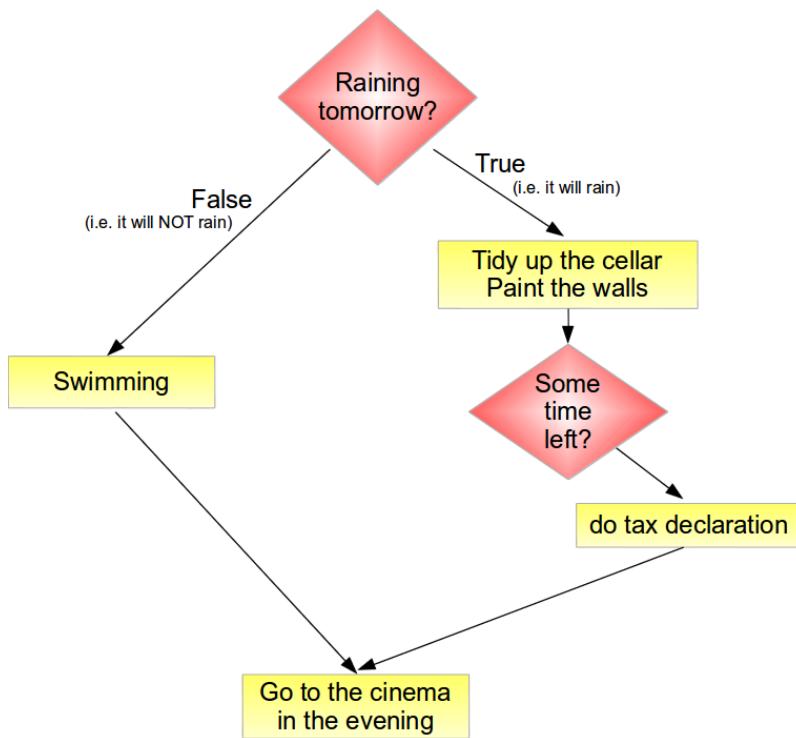
We can also graphically depict this. Such a workflow is often referred to in the programming environment as a so-called flowchart or program schedule (PAP):

To encode conditional statements in Python, we need to know how to combine statements into a block. So this seems to be the ideal moment to introduce the Python Block Principle in combination with the conditional statements.

COMBINING STATEMENTS INTO BLOCKS

In programming, blocks are used to treat sets of statements as if they were a single statement. A block consists of one or more statements. A program can be considered a block consisting of statements and other nested blocks. In programming languages, there exist various approaches to syntactically describing blocks: ALGOL 60 and Pascal, for example, use "begin" and "end", while C and similar languages use curly braces "{" and "}". Bash has another design that uses 'do ... done' and 'if ... fi' or 'case ... esac' constructs. All these languages have in common is that the indentation style is not binding. The problems that may arise from this will be shown in the next subchapter. You can safely skip this, if you are only interested in Python. For inexperienced programmers, this chapter may seem a bit difficult.

EXCURSION TO C

For all these approaches, there is a big drawback: The code may be fine for the interpreter or compiler of the language, but it can be written in a way that is poorly structured for humans. We want to illustrate this in the following "pseudo" C-like code snippet:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
```

```
    if (time_left)
            do_taxes();
} else
    enjoy_swimming();
go_cinema();
```

We will add a couple of spaces in front of calling 'go_cinema'. This puts this call at the same level as 'enjoy_swimming':

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
            do_taxes();
} else
    enjoy_swimming();
    go_cinema();
```

The execution of the program will not change: you will go to the cinema, whether it will be raining or not! However, the way the code is written falsely suggests that you will only go to the cinema if it does not rain. This example shows the possible ambiguity in the interpretation of C code by humans. In other words, you can inadvertently write the code, which leads to a procedure that does not "behave" as intended.
In this example lurks even another danger. What if the programmer had just forgotten to put the last statement out of the curly braces?

Should the code look like this:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
            do_taxes();
} else {
    enjoy_swimming();
    go_cinema();
}
```

In the following program you will only go to the cinema, if it is not a rainy day. That only makes sense if it's an open-air cinema.

The following code is the right code to go to the movies regardless of the weather:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
            do_taxes();
} else {
    enjoy_swimming();
```

```
    }
    go_cinema();}
```

The problem with C is that the way the code is indented has no meaning for the compiler, but it can lead to misunderstanding, if misinterpreted.

This is different in Python. Blocks are created with indentations, in other words, blocks are based on indentation. We can say "What you see is what you get!"
The example above might look like this in a Python program:

```
if raining_tomorrow:
    tidy_up_the_cellar()
    paint_the_walls()
    if time_left:
        do_taxes()
else:
    enjoy_swimming()
go_cinema()
```

There is no ambiguity in the Python version. The cinema visit of the couple will take place in any case, regardless of the weather. Moving the go_cinema() call to the same indentation level as enjoy_swimming() immediately changes the program logic. In that case, there will be no cinema if it rains.

We have just seen that it is useful to use indentations in C or C++ programs to increase the readability of a program. For the compilation or execution of a program they are but irrelevant. The compiler relies exclusively on the structuring determined by the brackets. Code in Python, on the other hand, can only be structured with indents. That is, Python forces the programmer to use indentation that should be used anyway to write nice and readable codes. Python does not allow you to obscure the structure of a program by misleading indentations.

Each line of a code block in Python must be indented with the same number of spaces or tabs. We will continue to delve into this in the next subsection on conditional statements in Python.

CONDITIONAL STATEMENTS IN PYTHON

The if statement is used to control the program flow in a Python program. This makes it possible to decide at runtime whether certain program parts should be executed or not.
The simplest form of an if statement in a Python program looks like this:

```
if condition:
    statement
    statement
    # further statements, if necessary
```

The indented block is only executed if the condition 'condition' is True. That is, if it is logically true.

The following example asks users for their nationality. The indented block will only be executed if the nationality is "french". If the user enters another nationality, nothing happens.

```python
person = input("Nationality? ")
if person == "french":
    print("Préférez-vous parler français?")
```

```
Nationality? french
Préférez-vous parler français?
```

Please note that if "French" is entered in the above program, nothing will happen. The check in the condition is case-sensitive and in this case only responds when "french" is entered in lower case. We now extend the condition with an "or"

```python
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
```

```
Nationality? french
Préférez-vous parler français?
```

The Italian colleagues might now feel disadvantaged because Italian speakers were not considered. We extend the program by another if statement.

```python
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
if person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
```

```
Nationality? italian
Preferisci parlare italiano?
```

This little script has a drawback. Suppose that the user enters "french" as a nationality. In this case, the block of the first "if" will be executed. Afterwards, the program checks if the second "if" can be executed as well. This doesn't make sense, because we know that the input "french" does not match the condition "italian" or "Italian". In other words: if the first 'if' matched, the second one can never match as well. In the end, this is an unnecessary check when entering "french" or "French".

We solve the problem with an "elif" condition. The 'elif' expression is only checked if the expression of a previous "elif" or "if" was false.

```python
person = input("Nationality? ")
```

```
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
elif person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
else:
    print("You are neither French nor Italian.")
    print("So, let us speak English!")
```

```
Nationality? german
You are neither French nor Italian.
So, let us speak English!
```

As in our example, most if statements also have "elif" and "else" branches. This means, there may be more than one "elif" branches, but only one "else" branch. The "else" branch must be at the end of the if statement. Other "elif" branches can not be attached after an 'else'. 'if' statements don't need either 'else' nor 'elif' statements.

The general form of the if statement in Python looks like this:

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2

...

elif another_condition:
    another_statement_block
else:
    else_block
```

If the condition "condition_1" is True, the statements of the block statement"_block_1" will be executed. If not, condition_2 will be evaluated. If "condition_2" evaluates to True, statement"_block_2" will be executed, if condition_2 is False, the other conditions of the following elif conditions will be checked, and finally if none of them has been evaluated to True, the indented block below the else keyword will be executed.

EXAMPLE WITH IF CONDITION

For children and dog lovers it is an interesting and frequently asked question how old their dog would be if it was not a dog, but a human being. To calculate this task, there are various scientific and pseudo-scientific approaches. An easy approach can be:

```
 A one-year-old dog is roughly equivalent to a 14-year-old human be
ing
 A dog that is two years old corresponds in development to a 22 yea
r old person.
 Each additional dog year is equivalent to five human years.
```

The following example program in Python requires the age of the dog as the input and calculates the age in human years according to the above rule. 'input' is a statement, where the program flow stops and waits for user input printing out the message in brackets:

```python
age = int(input("Age of the dog: "))
print()
if age < 0:
    print("This cannot be true!")
elif age == 0:
    print("This corresponds to 0 human years!")
elif age == 1:
    print("Roughly 14 years!")
elif age == 2:
    print("Approximately 22 years!")
else:
    human = 22 + (age -2) * 5
    print("Corresponds to  " + str(human) + " human years!")

Age of the dog: 1

Roughly 14 years!
```

In [ ]:
```python
We will use this example again in our tutorial
```

Using if statements within programs can easily lead to complex decision trees, i.e. every if statements can be seen like the branches of a tree.

We will read in three float numbers in the following program and will print out the largest value:

```python
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y and x > z:
    maximum = x
elif y > x and y > z:
    maximum = y
else:
    maximum = z

print("The maximal value is: " + str(maximum))
```

```
1st Number: 1
2nd Number: 2
3rd Number: 3
The maximal value is: 3.0
```

There are other ways to write the conditions like the following one:

```python
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y:
    if x > z:
        maximum = x
    else:
        maximum = z
else:
    if y > z:
        maximum = y
    else:
        maximum = z

print("The maximal value is: " + str(maximum))
```

```
1st Number: 4.4
2nd Number: 1
3rd Number: 8.3
The maximal value is: 8.3
```

Another way to find the maximum can be seen in the following example. We are using the built-in function max, which calculates the maximum of a list or a tuple of numerical values:

```python
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

maximum = max((x,y,z))

print("The maximal value is: " + str(maximum))
```

```
1st Number: 1
2nd Number: 4
3rd Number: 5.5
The maximal value is: 5.5
```

TERNARY IF STATEMENT

Let's look at the following English sentence, which is valid in Germany :

The maximum speed is 50 if we are within the city, otherwise it is 100.

This can be translated into ternary Python statements:

```python
inside_city_limits = True
maximum_speed = 50 if inside_city_limits else 100
print(maximum_speed)
```

```
50
```

This code looks like an abbreviation for the following code:

```python
im_ort = True
if im_ort:
    maximale_geschwindigkeit = 50
else:
    maximale_geschwindigkeit = 100
print(maximale_geschwindigkeit)
```

```
50
```

But it is something fundamentally different. "50 if inside_city_limits else 100" is an expression that we can also use in function calls.

EXERCISES

EXERCISE 1

- A leap year is a calendar year containing an additional day added to keep the calendar year synchronized with the astronomical or seasonal year. In the Gregorian calendar, each leap year has 366 days instead of 365, by extending February to 29 days rather than the common 28. These extra days occur in years which are multiples of four (with the exception of centennial years not divisible by 400). Write a Python program, which asks for a year and calculates, if this year is a leap year or not.

- Body mass index (BMI) is a value derived from the mass (weight) and height of a person. The BMI is defined as the body mass divided by the square of the body height, and is universally

expressed in units of kg/m2, resulting from mass in kilograms and height in metres. $BMI = \dfrac{m}{l^2}$

Write a program, which asks for the length and the weight of a person and returns an evaluation string according to the following table:

Category	BMI (kg/m²)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

SOLUTIONS

EXERCISE 1:

We will use the modulo operator in the following solution. 'n % m' returns the remainder, if you divide (integer division) n by m. '5 % 3' returns 2 for example.

```python
year = int(input("Which year? "))

if year % 4:
    # not divisible by 4
    print("no leap year")
elif year % 100 == 0 and year % 400 != 0:
    print("no leap year")
else:
    print("leap year")
```

```
Which year? 2020
leap year
```

```python
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

```
What is your height? 180
What is your weight? 74
0.0022839506172839510
Very severely underweight
```

FOOTNOTES

[1] LOOP is a programming language without conditionals, but this language is purely pedagogical. It has been designed by the German computer scientist Uwe Schöning. The only operations which are supported by LOOP are assignments, additions and loopings. But LOOP is of no practical interest and besides this, it is only a proper subset of the computable functions.

LOOPS

GENERAL STRUCTURE OF A LOOP

Many algorithms make it necessary for a programming language to have a construction which makes it possible to carry out a sequence of statements repeatedly. The code within the loop, i.e. the code carried out repeatedly, is called the body of the loop.
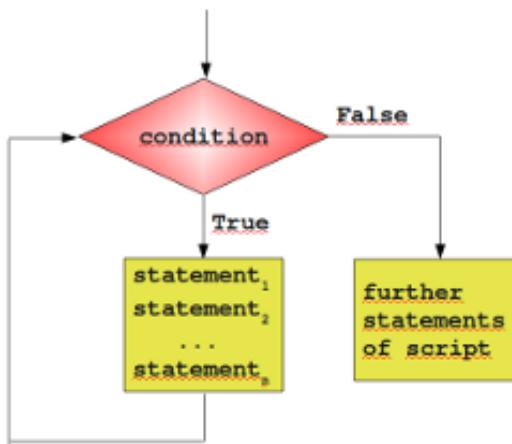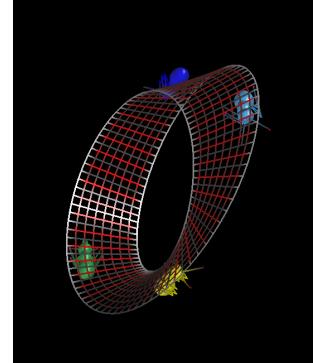
Essentially, there are three different kinds of loops:

- Count-controlled loops A construction for repeating a loop a certain number of times. An example of this kind of loop is the for-loop of the programming language C:

  for (i=0; i <= n; i++)

  Python doesn't have this kind of loop.

- Condition-controlled loop A loop will be repeated until a given condition changes, i.e. changes from True to False or from False to True, depending on the kind of loop. There are 'while loops' and 'do while' loops with this behaviour.

- Collection-controlled loop This is a special construct which allows looping through the elements of a 'collection', which can be an array, list or other ordered sequence. Like the for loop of the bash shell (e.g. for i in *, do echo $i; done) or the foreach loop of Perl.

Python supplies two different kinds of loops: the while loop and the for loop, which correspond to the condition-controlled loop and collection-controlled loop.

Most loops contain a counter or more generally, variables, which change their values in the course of calculation. These variables have to be initialized before the loop is started. The counter or other variables,

which can be altered in the body of the loop, are contained in the condition. Before the body of the loop is executed, the condition is evaluated. If it evaluates to False, the while loop is finished. In other words, the program flow will continue with the first statement after the while statement, i.e. on the same indentation level as the while loop. If the condition is evaluated to True, the body, - the indented block below the line with "while" - gets executed. After the body is finished, the condition will be evaluated again. The body of the loop will be executed as long as the condition yields True.

A SIMPLE EXAMPLE WITH A WHILE LOOP

It's best to illustrate the operating principle of a loop with a simple Python example. The following small script calculates the sum of the numbers from 1 to 100. We will later introduce a more elegant way to do it.

```python
n = 100

total_sum = 0
counter = 1
while counter <= n:
    total_sum += counter
    counter += 1

print("Sum of 1 until " + str(n) +  " results in " + str(total_su
m))
```

```
Sum of 1 until 100 results in 5050
```

EXERCISE

Write a program, which asks for the initial balance K0 and for the interest rate. The program shall calculate the new capital K1 after one year including the interest.

Extend the program with a while-loop, so that the capital Kn after a period of n years can be calculated.

```python
K = float(input("Starting capital? "))
p = float(input("Interest rate? "))
n = int(input("Number of years? "))

i = 0
while i < n:
    K +=  K * p / 100.0
    # or K *=  1 +  p / 100.0
    i += 1
    print(i, K)
print("Capital after " + str(n) + " ys: " + str(K))
```
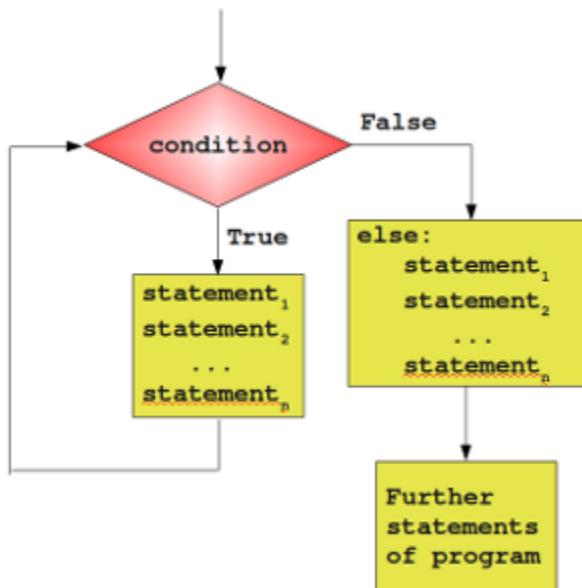
```
1 100.6
2 101.2036
3 101.8108216
4 102.4216865296
5 103.0362166487776
6 103.65443394867026
7 104.27636055236228
8 104.90201871567645
9 105.53143082797051
10 106.16461941293834
Capital after 10 ys: 106.16461941293834
```

THE ELSE PART

Similar to the if statement, the while loop of Python has also an optional else part. This is an unfamiliar construct for many programmers of traditional programming languages. The statements in the else part are executed, when the condition is not fulfilled anymore. Some might ask themselves now, where the possible benefit of this extra branch is. If the statements of the additional else part were placed right after the while loop without an else, they would have been executed anyway, wouldn't they? We need to understand a new language construction, i.e. the break statement, to obtain a benefit from the additional else branch of the while loop. The general syntax of a while loop looks like this:

```
while condition:
    statement_1
    ...
    statement_n
else:
    statement_1
```

```
        ...
        statement_n


    while potatoes:
        peel

        cut
    else:
        pot pot on oven
        cook for 40 minutes

    wash salad
    and so on
```
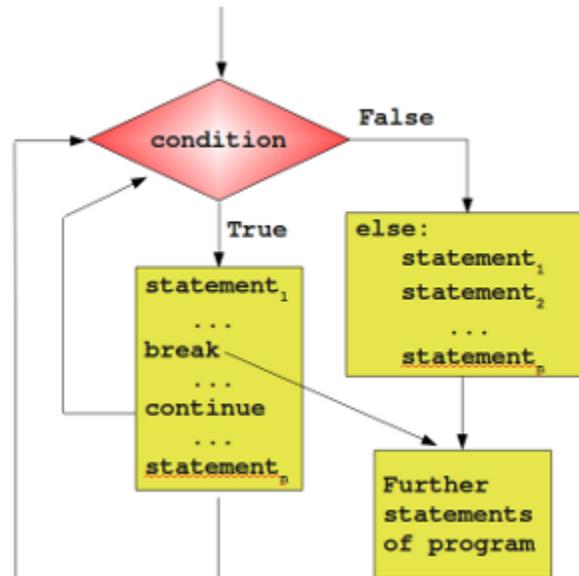
PREMATURE TERMINATION OF A WHILE LOOP

So far, a while loop only ends, if the condition in the loop head is fulfilled. With the help of a break statement a while loop can be left prematurely, i.e. as soon as the control flow of the program comes to a break inside of a while loop (or other loops) the loop will be immediately left. "break" shouldn't be confused with the continue statement. "continue" stops the current iteration of the loop and starts the next iteration by checking the condition. Here comes the crucial point: If a loop is left by break, the else part is not executed.

This behaviour will be illustrated in the following example, a little guessing number game. A human player has to guess a number between a range of 1 to n. The player inputs his guess. The program informs the player, if this number is larger, smaller or equal to the secret number, i.e. the number which the program has randomly created. If the player wants to gives up, he or she can input a 0 or a negative number. Hint: The program needs to create a random number. Therefore it's necessary to include the module "random".


```python
import random
upper_bound = 20
lower_bound = 1
#to_be_guessed = int(n * random.random()) + 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
```

```python
    if guess > 0:
        if guess > to_be_guessed:
            print("Number too large")
        elif guess < to_be_guessed:
            print("Number too small")
    else:
        # 0 means giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
else:
    print("Congratulations. You made it!")
```

```
Number too small

Number too large

Number too large

Number too large

Congratulations. You made it!
```

The previous program doesn't check if the `number` makes sense, i.e. if the number is between the boundaries `upper_bound` and `over_bound`. We can improve our program:

```python
import random
upper_bound = 20
lower_bound = 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:   # giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess > to_be_guessed:
        print("Number too large")
    elif guess < to_be_guessed:
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
Congratulations. You made it!
```

The boundaries should be adapted according to the user input:

```python
import random
upper_bound = 20
lower_bound = 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:    # giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess > to_be_guessed:
        upper_bound = guess - 1
        print("Number too large")
    elif guess < to_be_guessed:
        lower_bound = guess + 1
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
Number too small

Number too small

Number too large

Number too small

Congratulations. You made it!
```

```python
import random
upper_bound = 20
lower_bound = 1
adaptive_upper_bound = upper_bound
adaptive_lower_bound = lower_bound
#to_be_guessed = int(n * random.random()) + 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:    # giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
```

```python
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess < adaptive_lower_bound or guess > adaptive_upper_bound:
        print("Your guess is contradictory to your previous guesses!")
    elif guess > to_be_guessed:
        adaptive_upper_bound = guess - 1
        print("Number too large")
    elif guess < to_be_guessed:
        adaptive_lower_bound = guess + 1
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
Number too small

Your guess is contradictory to your previous guesses!

Number too small

Your guess is contradictory to your previous guesses!

guess not within boundaries!

Number too small

Number too small

Number too small

Number too small

Congratulations. You made it!
```

FOR LOOPS

INTRODUCTION

Like the while loop the for loop is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times.

There are hardly any programming languages without for loops, but the for loop exists in many different flavours, i.e. both the syntax and the semantics differs from one programming language to another.

Different kinds of for loops:

- Count-controlled for loop (Three-expression for loop)

  This is by far the most common type. This statement is the one used by C. The header of this kind of for loop consists of a three-parameter loop control expression. Generally it has the form: for (A; Z; I) A is the initialisation part, Z determines a termination expression and I is the counting expression, where the loop variable is incremented or dcremented. An example of this kind of loop is the for-loop of the programming language C: for (i=0; i <= n; i++) This kind of for loop is not implemented in Python!

- Numeric Ranges
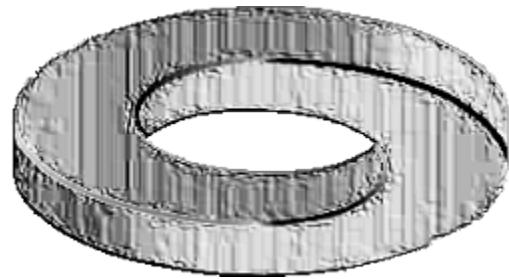
  This kind of for loop is a simplification of the previous kind. It's a counting or enumerating loop. Starting with a start value and counting up to an end value, like for i = 1 to 100 Python doesn't use this either.

- Vectorized for loops

  They behave as if all iterations are executed in parallel. That is, for example, all expressions on the right side of assignment statements get evaluated before the assignments.

- Iterator-based for loop

  Finally, we come to the one used by Python. This kind of for loop iterates over an enumeration of a set of items. It is usually characterized by the use of an implicit or explicit iterator. In each iteration step a loop variable is set to a value in a sequence or other data collection. This kind of for loop is known in most Unix and Linux shells and it is the one which is implemented in Python.

SYNTAX OF THE FOR LOOP

As we mentioned earlier, the Python for loop is an iterator based for loop. It steps through the items of lists,

tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

for in : else:

The items of the sequence object are assigned one after the other to the loop variable; to be precise the variable points to the items. For each item the loop body is executed.

Example of a simple for loop in Python:

```
languages = ["C", "C++", "Perl", "Python"]
for language in languages:
    print(language)
```

```
C
C++
Perl
Python
```

The else block is special; while Perl programmer are familiar with it, it's an unknown concept to C and C++ programmers. Semantically, it works exactly as the optional else of a while loop. It will be executed only if the loop hasn't been "broken" by a break statement. So it will only be executed, after all the items of the sequence in the header have been used.

If a break statement has to be executed in the program flow of the for loop, the loop will be exited and the program flow will continue with the first statement following the for loop, if there is any at all. Usually break statements are wrapped into conditional statements, e.g.

```
edibles = ["bacon", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious bacon
No more spam please!
Finally, I finished stuffing myself
```

Removing "spam" from our list of edibles, we will gain the following output:

```
$ python for.py
Great, delicious bacon
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
$
```

Maybe, our disgust with spam is not so high that we want to stop consuming the other food. Now, this calls the continue statement into play . In the following little script, we use the *continue* statement to go on with our list of edibles, when we have encountered a spam item. So *continue* prevents us from eating spam!

```python
edibles = ["bacon", "spam", "eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)

print("Finally, I finished stuffing myself")
```

```
Great, delicious bacon
No more spam please!
Great, delicious eggs
Great, delicious nuts
Finally, I finished stuffing myself
```

THE RANGE() FUNCTION

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions: Example:

```python
range(5)
```

Output:
```
range(0, 5)
```

This result is not self-explanatory. It is an object which is capable of producing the numbers from 0 to 4. We can use it in a for loop and you will see what is meant by this:

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

```
0
1
2
3
4
```

range(n) generates an iterator to progress the integer numbers starting with 0 and ending with (n -1). To produce the list with these numbers, we have to cast range() with the list(), as we do in the following example.

```
list(range(10))
```

Output: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

range() can also be called with two arguments:

```
range(begin, end)
```

The above call produces the list iterator of numbers starting with `begin` (inclusive) and ending with one less than the number `end`.

Example:

```
range(4, 10)
```

Output: `range(4, 10)`

```
list(range(4, 10))
```

Output: `[4, 5, 6, 7, 8, 9]`

So far the increment of range() has been 1. We can specify a different increment with a third argument. The increment is called the `step`. It can be both negative and positive, but not zero:

```
range(begin,end, step)
```

Example with step:

```
list(range(4, 50, 5))
```

Output: `[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]`

It can be done backwards as well:

```
list(range(42, -12, -7))
```

Output: `[42, 35, 28, 21, 14, 7, 0, -7]`

The range() function is especially useful in combination with the for loop, as we can see in the following example. The range() function supplies the numbers from 1 to 100 for the for loop to calculate the sum of these numbers:

```
n = 100

sum = 0
for counter in range(1, n+1):
    sum = sum + counter

print("Sum of 1 until %d: %d" % (n, sum))
```
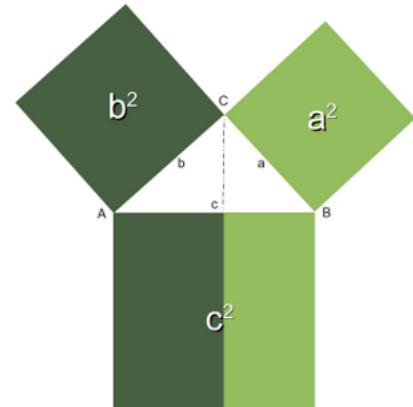
```
Sum of 1 until 100: 5050
```

CALCULATION OF THE PYTHAGOREAN NUMBERS

Generally, it is assumed that the Pythagorean theorem was discovered by Pythagoras that is why it has his name. However, there is a debate whether the Pythagorean theorem might have been discovered earlier or by others independently. For the Pythagoreans, - a mystical movement, based on mathematics, religion and philosophy, - the integer numbers satisfying the theorem were special numbers, which had been sacred to them.

These days Pythagorean numbers are not mystical anymore. Though to some pupils at school or other people, who are not on good terms with mathematics, they may still appear so.

So the definition is very simple: Three integers satisfying $a2+b2=c2$ are called Pythagorean numbers.

The following program calculates all pythagorean numbers less than a maximal number. Remark: We have to import the math module to be able to calculate the square root of a number.

In [ ]:

```
from math import sqrt
n = int(input("Maximal Number? "))
for a in range(1, n+1):
    for b in range(a, n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print(a, b, c)
```

ITERATING OVER LISTS WITH RANGE()

If you have to access the indices of a list, it doesn't seem to be a good idea to use the for loop to iterate over the lists. We can access all the elements, but the index of an element is not available. However, there is a way to access both the index of an element and the element itself. The solution lies in using range() in combination with the length function len():

```python
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
print()
```

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
```

Remark: If you apply len() to a list or a tuple, you get the number of elements of this sequence.

LIST ITERATION WITH SIDE EFFECTS

If you loop over a list, it's best to avoid changing the list in the loop body. Take a look at the following example:

```python
colours = ["red"]
for i in colours:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)
```

```
['red', 'black', 'white']
```

To avoid these side effects, it's best to work on a copy by using the slicing operator, as can be seen in the next example:

```python
colours = ["red"]
```

```
for i in colours[:]:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)
```

```
['red', 'black']
```

We still might have done something, we shouldn't have done. We changed the list "colours", but our change didn't have any effect on the loop. The elements to be looped remained the same during the iterations.

ITERATORS AND ITERABLES

The Python forums and other question-and-answer websites like Quora and Stackoverflow are full of questions concerning 'iterators' and 'iterable'. Some want to know how they are defined and others want to know if there is an easy way to check, if an object is an iterator or an iterable. We will provide a function for this purpose.

We have seen that we can loop or iterate over various Python objects like lists, tuples and strings. For example:

```python
for city in ["Berlin", "Vienna", "Zurich"]:
    print(city)
for language in ("Python", "Perl", "Ruby"):
    print(city)
for char in "Iteration is easy":
    print(char)
```

```
Berlin
Vienna
Zurich
Python
Perl
Ruby
I
t
e
r
a
t
i
o
n

i
s

e
a
s
y
```

This form of looping can be seen as iteration. Iteration is not restricted to explicit for loops. If you call the function sum, - e.g. on a list of integer values, - you do iteration as well.

So what is the difference between an iterable and an iterator?

On one hand, they are the same: You can iterate with a for loop over iterators and iterables. Every iterator is also an iterable, but not every iterable is an iterator. E.g. a list is iterable but a list is not an iterator! An iterator can be created from an iterable by using the function 'iter'. To make this possible the class of an object needs either a method '`__iter__`', which returns an iterator, or a '`__getitem__`' method with sequential indexes starting with 0.

Iterators are objects with a '`__next__`' method, which will be used when the function 'next()' is called.

So what is going on behind the scenes, when a for loop is executed? The for statement calls iter() on the object (which should be a so-called container object), over which it is supposed to loop . If this call is successful, the iter call will return an iterator object that defines the method `__next__`() which accesses elements of the object one at a time. The `__next__`() method will raise a StopIteration exception, if there are no further elements available. The for loop will terminate as soon as it catches a StopIteration exception. You can call the `__next__`() method using the next() built-in function. This is how it works:

```python
cities = ["Berlin", "Vienna", "Zurich"]
iterator_obj = iter(cities)
print(iterator_obj)

print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
```

```
<list_iterator object at 0x0000016FBDAEEC88>
Berlin
Vienna
Zurich
```

If we called 'next(iterator_obj)' one more time, it would return 'StopIteration'

The following function 'iterable' will return True, if the object 'obj' is an iterable and False otherwise.

```python
def iterable(obj):
    try:
        iter(obj)
        return True
    except TypeError:
        return False

for element in [34, [4, 5], (4, 5), {"a":4}, "dfsdf", 4.5]:
    print(element, "iterable: ", iterable(element))
```

```
34 iterable:  False
[4, 5] iterable:  True
(4, 5) iterable:  True
{'a': 4} iterable:  True
dfsdf iterable:  True
4.5 iterable:  False
```

We have described how an iterator works. So if you want to add an iterator behavior to your class, you have to add the __iter__ and the __next__ method to your class. The __iter__ method returns an iterator object. If the class contains a __next__ , it is enough for the __iter__ method to return self, i.e. a reference to itself:

```python
class Reverse:
    """
    Creates Iterators for looping over a sequence backwards.
    """

    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

lst = [34, 978, 42]
lst_backwards = Reverse(lst)
for el in lst_backwards:
    print(el)
```

```
42
978
34
```

PRINT

INTRODUCTION

In principle, every computer program has to communicate with the environment or the "outside world". For this purpose nearly every programming language has special I/O functionalities, i.e. input/output. This ensures the interaction or communication with the other components e.g. a database or a user. Input often comes - as we have already seen - from the keyboard and the corresponding Python command, or better, the corresponding Python function for reading from the standard input is input().

We have also seen in previous examples of our tutorial that we can write into the standard output by using print. In this chapter of our tutorial we want to have a detailed look at the print function. As some might have skipped over it, we want to emphasize that we wrote "print function" and not "print statement". You can easily find out how crucial this difference is, if you take an arbitrary Python program written in version 2.x and if you try to let it run with a Python3 interpreter. In most cases you will receive error messages. One of the most frequently occurring errors will be related to print, because most programs contain prints. We can generate the most typical error in the interactive Python shell:

```
print 42
  File "", line 1
    print 42

  File "<ipython-input-1-5d54469f7ddf>", line 1
    print 42
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print(42)?
```

This is a familiar error message for most of us: We have forgotten the parentheses. "print" is - as we have already mentioned - a function in version 3.x. Like any other function print expects its arguments to be surrounded by parentheses. So parentheses are an easy remedy for this error:

```
print(42)
```

```
42
```

But this is not the only difference to the old print. The output behaviour has changed as well.

PRINT FUNCTION

The arguments of the print function are the following ones:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=Fals
```

e)

The print function can print an arbitrary number of values ("value1, value2, ..."), which are separated by commas. These values are separated by a space in the output. In the following example, we can see two print calls. We are printing two values in both cases, i.e. a string and a float number:

```
a =  3.564
print("a = ", a)

a =  3.564

print("a = \n", a)

a =
 3.564
```

We can learn from the second print of the example that a blank between two values, i.e. "a = \textbackslash n" and "3.564", is always printed, even if the output is continued in the following line. This is different from Python 2, as there will be no blank printed, if a new line has been started. It's possible to redefine the seperator between values by assigning an arbitrary string to the keyword parameter "sep", i.e. an empty string or a smiley:

```
print("a","b")

a b

print("a","b",sep="")

ab

print(192,168,178,42,sep=".")

192.168.178.42

print("a","b",sep=":-)")

a:-)b
```

A print call is ended by a newline, as we can see in the following usage:

```
for i in range(4):
    print(i)

0
1
2
3
```

To change this behaviour, we can assign an arbitrary string to the keyword parameter "end". This string will be used for ending the output of the values of a print call:

```
for i in range(4):
    print(i, end=" ")
```

0 1 2 3

```
for i in range(4):
    print(i, end=" :-) ")
```

0 :-) 1 :-) 2 :-) 3 :-)

The output of the print function is send to the standard output stream (sys.stdout) by default. By redefining the keyword parameter "file" we can send the output into a different stream e.g. sys.stderr or a file:

```
fh = open("data.txt","w")
print("42 is the answer, but what is the question?", file=fh)
fh.close()
```

We can see that we don't get any output in the interactive shell. The output is sent to the file "data.txt". It's also possible to redirect the output to the standard error channel this way:

```
import sys
# output into sys.stderr:

print("Error: 42", file=sys.stderr)
```

Error: 42

FORMATTED OUTPUT

MANY WAYS FOR A NICER OUTPUT

In this chapter of our Python tutorial we will have a closer look at the various ways of creating nicer output in Python. We present all the different ways, but we recommend that you should use the format method of the string class, which you will find at end of the chapter. "string format" is by far the most flexible and Pythonic approach.

So far we have used the print function in two ways, when we had to print out more than two values:

The easiest way, but not the most elegant one:

We used print with a comma separated list of values to print out the results, as we can see in the following example. All the values are separated by blanks, which is the default behaviour. We can change the default value to an arbitrary string, if we assign this string to the keyword parameter "sep" of the print function:

```
q = 459
p = 0.098
print(q, p, p * q)
```

```
459 0.098 44.982
```

```
print(q, p, p * q, sep=",")
```

```
459,0.098,44.982
```

```
print(q, p, p * q, sep=" :-) ")
```

```
459 :-) 0.098 :-) 44.982
```

Alternatively, we can construe a new string out of the values by using the string concatenation operator:

```
print(str(q) + " " + str(p) + " " + str(p * q))
```
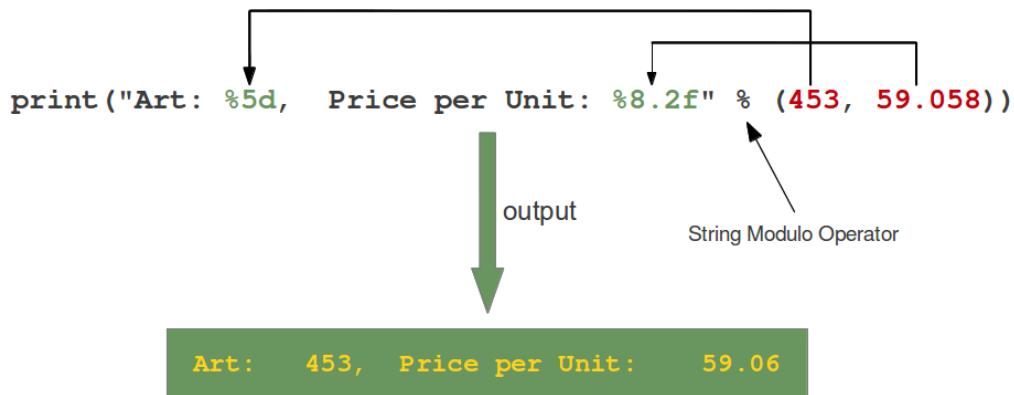
```
459 0.098 44.982
```

The second method is inferior to the first one in this example.
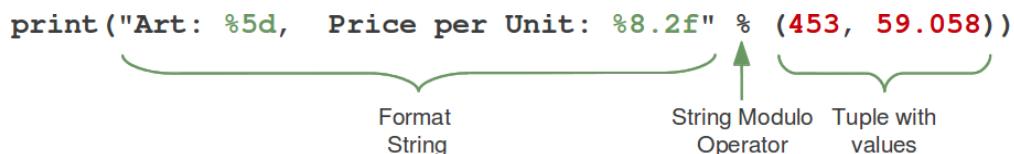
THE OLD WAY OR THE NON-EXISTING PRINTF AND SPRINTF

Is there a printf in Python? A burning question for Python newbies coming from C, Perl, Bash or other programming languages who have this statement or function. To answer "Python has a print function and no printf function" is only one side of the coin or half of the truth. One can go as far as to say that this answer is not true. So there is a "printf" in Python? No, but the functionality of the "ancient" printf is contained in Python. To this purpose the modulo operator "%" is overloaded by the string class to perform string formatting. Therefore, it is often called string modulo (or somethimes even called modulus) operator, though it has not a lot in common with the actual modulo calculation on numbers. Another term for it is "string interpolation", because it interpolates various class types (like int, float and so on) into a formatted string. In many cases the string created via the string interpolation mechanism is used for outputting values in a special way. But it can also be used, for example, to create the right format to put the data into a database. Since Python 2.6 has been introduced, the string method format should be used instead of this old-style formatting. Unfortunately, string modulo "%" is still available in Python3 and what is even worse, it is still widely used. That's why we cover it in great detail in this tutorial. You should be capable of understanding it, when you encounter it in some Python code. However, it is very likely that one day this old style of formatting will be removed from the language. So you should get used to str.format().

The following diagram depicts how the string modulo operator works:

On the left side of the "string modulo operator" there is the so-called format string and on the right side is a tuple with the content, which is interpolated in the format string. The values can be literals, variables or arbitrary arithmetic expressions.

The format string contains placeholders. There are two of those in our example: "%5d" and "%8.2f".
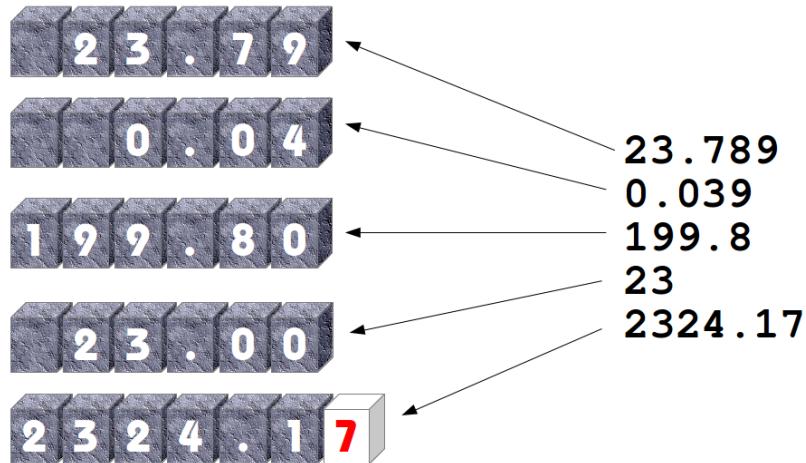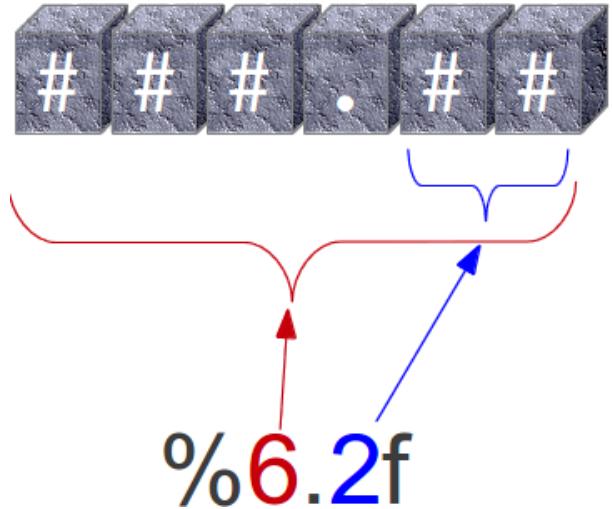
The general syntax for a format placeholder is

```
%[flags][width][.precision]type
```

Let's have a look at the placeholders in our example. The second one "%8.2f" is a format description for a float number. Like other placeholders, it is introduced with the "%" character. This is followed by the total number of digits the string should contain. This number includes the decimal point and all the digits, i.e. before and after the decimal point. Our float number 59.058 has to be formatted with 8 characters. The decimal part of the number or the precision is set to 2, i.e. the number following the "." in our placeholder. Finally, the last character "f" of our placeholder stands for "float".

If you look at the output, you will notice that the 3 decimal digits have been rounded. Furthermore, the number has been preceded in the output with 3 leading blanks.

The first placeholder "%5d" is used for the first component of our tuple, i.e. the integer 453. The number will be printed with 5 characters. As 453 consists only of 3 digits, the output is padded with 2 leading blanks. You may have expected a "%5i" instead of "%5d". Where is the difference? It's easy: There is no difference between "d" and "i" both are used for formatting integers. The advantage or beauty of a formatted output can only be seen, if more than one line is printed with the same pattern. In the following picture you can see, how it looks, if five float numbers are printed with the placeholder "%6.2f" in subsequent lines:

23.789
0.039
199.8
23
2324.17

Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Obsolete and equivalent to 'd', i.e. signed integer decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using `repr()`).
s	String (converts any python object using `str()`).
%	No argument is converted, results in a "%" character in the result.

The following examples show some example cases of the conversion rules from the table above:

```python
print("%10.3e"% (356.08977))
```

```
  3.561e+02
```

```python
print("%10.3E"% (356.08977))
```

```
  3.561E+02
```

```python
print("%10o"% (25))
```

```
        31
```

```python
print("%10.3o"% (25))
```

```
       031
```

```python
print("%10.5o"% (25))
```

```
     00031
```

```python
print("%5x"% (47))
```

```
   2f
```

```python
print("%5.4x"% (47))
```

```
 002f
```

```python
print("%5.4X"% (47))
```

```
 002F
```

```python
print("Only one percentage sign: %% " % ())
```

```
Only one percentage sign: %
```

Examples:

```
 0X2F
```

```
   2F
```

```
0X002F
```

```
 0o31
```

```
+42
```

```
 42
```

```
+42
```

```
 42
```

Even though it may look so, the formatting is not part of the print function. If you have a closer look at our examples, you will see that we passed a formatted string to the print function. Or to put it in other words: If the string modulo operator is applied to a string, it returns a string. This string in turn is passed in our examples to the print function. So, we could have used the string modulo functionality of Python in a two layer approach as well, i.e. first create a formatted string, which will be assigned to a variable and this variable is passed to the print function:

```
Price: $    356.09
```

THE PYTHONIC WAY: THE STRING METHOD "FORMAT"

The Python help function is not very helpful concerning the string format method. All it says is this:

```
|  format(...)
|      S.format(*args, **kwargs) -> str
|
|      Return a formatted version of S, using substitutions from a
rgs and kwargs.
|      The substitutions are identified by braces ('{' and '}').
|
```

Let's dive into this topic a little bit deeper: The format method was added in Python 2.6. The general form of this method looks like this:
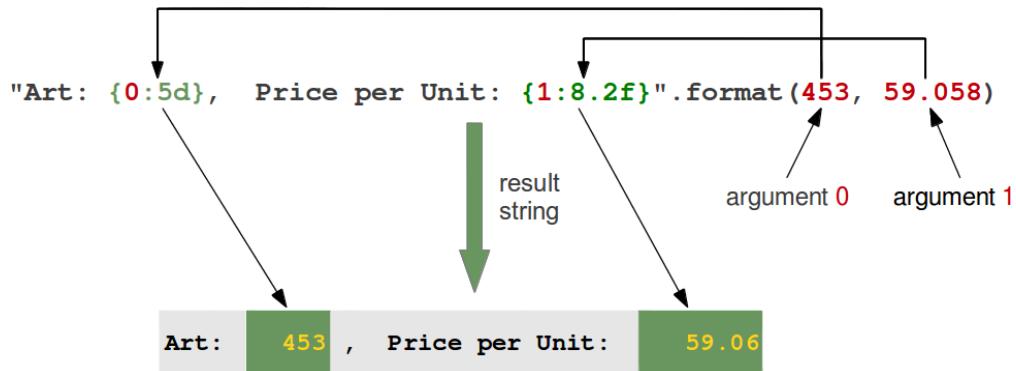
```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

The template (or format string) is a string which contains one or more format codes (fields to be replaced) embedded in constant text. The "fields to be replaced" are surrounded by curly braces {}. The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments, according to the rules which we will specify soon. Anything else, which is not contained in curly braces will be literally printed, i.e. without any changes. If a brace character has to be printed, it has to be escaped by doubling it: {{ and }}.

There are two kinds of arguments for the .format() method. The list of arguments starts with zero or more positional arguments (p0, p1, ...), it may be followed by zero or more keyword arguments of the form name=value.

A positional parameter of the format method can be accessed by placing the index of the parameter after the opening brace, e.g. {0} accesses the first parameter, {1} the second one and so on. The index inside of the curly braces can be followed by a colon and a format string, which is similar to the notation of the string modulo, which we had discussed in the beginning of the chapter of our tutorial, e.g. {0:5d} If the positional parameters are used in the order in which they are written, the positional argument specifiers inside of the braces can be omitted, so '{} {} {}' corresponds to '{0} {1} {2}'. But they are needed, if you want to access them in different orders: '{2} {1} {0}'.

The following diagram with an example usage depicts how the string method "format" works works for positional parameters:

Examples of positional parameters:

Output: `'First argument: 47, second one: 11'`

Output: `'Second argument: 11, first one: 47'`
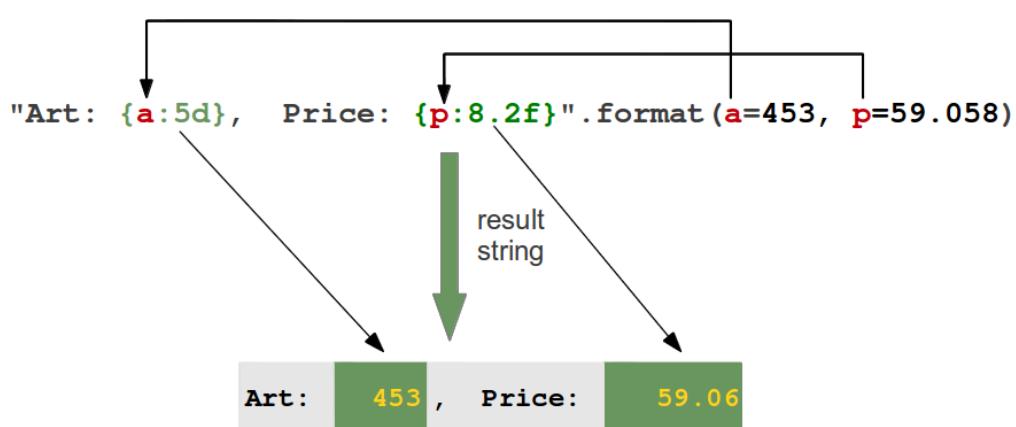
Output: `'Second argument:  11, first one:   47.42'`

Output: `'First argument: 47, second one: 11'`

Output: `'various precisions:   1.41 or  1.415'`

In the following example we demonstrate how keyword parameters can be used with the format method:

Output: `'Art:   453,  Price:    59.06'`

It's possible to "left or right justify" data with the format method. To this end, we can precede the formatting with a "<" (left justify) or ">" (right justify). We demonstrate this with the following examples:

Output: `'Spam & Eggs:                  6.99'`

Output: `'                 Spam & Eggs:   6.99'`

Flag	Meaning
#	Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively.
0	The conversion result will be zero padded for numeric values.
-	The converted value is left adjusted
	If no sign (minus sign e.g.) is going to be written, a blank space is inserted before the value.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

Option	Meaning
`'<'`	The field will be left-aligned within the available space. This is usually the default for strings.
`'>'`	The field will be right-aligned within the available space. This is the default for numbers.

`'0'`	If the width field is preceded by a zero ('0') character, sign-aware zero-padding for numeric types will be enabled.
	
	`>>> x = 378`
	`>>> print("The value is {:06d}".format(x))`
	`The value is 000378`
	`>>> x = -378`
	`>>> print("The value is {:06d}".format(x))`
	`The value is -00378`

`','`	This option signals the use of a comma for a thousands separator.
	
	`>>> print("The value is {:,}".format(x))`
	`The value is 78,962,324,245`
	`>>> print("The value is {0:6,d}".format(x))`
	`The value is 5,897,653,423`
	`>>> x = 5897653423.89676`
	`>>> print("The value is {0:12,.3f}".format(x))`
	`The value is 5,897,653,423.897`

`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types.

Option	Meaning
'^'	Forces the field to be centered within the available space.

Unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

Additionally, we can modify the formatting with the sign option, which is only valid for number types:

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers, which is the default behavior.
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

USING DICTIONARIES IN "FORMAT"

We have seen in the previous chapters that we have two ways to access the values to be formatted:

Using the position or the index:

```
print("The capital of {0:s} is {1:s}".format("Ontario","Toronto"))

The capital of Ontario is Toronto
```

Just to mention it once more: We could have used empty curly braces in the previous example! Using keyword parameters:

```
print("The capital of {province} is {capital}".format(province="On
tario",capital="Toronto"))

The capital of Ontario is Toronto
```

The second case can be expressed with a dictionary as well, as we can see in the following code:

```
data = dict(province="Ontario",capital="Toronto")
data
```
Output:
```
{'province': 'Ontario', 'capital': 'Toronto'}
```

```
print("The capital of {province} is {capital}".format(**data))
```

```
The capital of Ontario is Toronto
```

The double "*" in front of data turns data automatically into the form 'province="Ontario",capital="Toronto"'. Let's look at the following Python program:

```python
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
                   "Austria" : "Vienna",
                   "Netherlands" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    print("{country}: {capital}".format(country=c, capital=capita
l_country[c]))
```

```
Countries and their capitals:
United States: Washington
US: Washington
Canada: Ottawa
Germany: Berlin
France: Paris
England: London
UK: London
Switzerland: Bern
Austria: Vienna
Netherlands: Amsterdam
```

We can rewrite the previous example by using the dictionary directly. The output will be the same:

```python
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
```

```
                    "Austria" : "Vienna",
                    "Netherlands" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    format_string = c + ": {" + c + "}"
    print(format_string.format(**capital_country))
```

```
Countries and their capitals:
United States: Washington
US: Washington
Canada: Ottawa
Germany: Berlin
France: Paris
England: London
UK: London
Switzerland: Bern
Austria: Vienna
Netherlands: Amsterdam
```

USING LOCAL VARIABLE NAMES IN "FORMAT"

"locals" is a function, which returns a dictionary with the current scope's local variables, i.e- the local variable names are the keys of this dictionary and the corresponding values are the values of these variables:

```
a = 42
b = 47
def f():
    return 42
locals()
```

The dictionary returned by locals() can be used as a parameter of the string format method. This way we can use all the local variable names inside of a format string.

Continuing with the previous example, we can create the following output, in which we use the local variables a, b and f:

```
print("a={a}, b={b} and f={f}".format(**locals()))
```

```
a=42, b=47 and f=<function f at 0x00000285FFA729D8>
```

OTHER STRING METHODS FOR FORMATTING

The string class contains further methods, which can be used for formatting purposes as well: ljust, rjust, center and zfill.

Let S be a string, the 4 methods are defined like this:

- center(...):

```
S.center(width[, fillchar]) -> str
```

Return S centred in a string of length width. Padding is done using the specified fill character. The default value is a space.

```
Examples:
```

```
s = "Python"
s.center(10)
```

Output: ' Python '

```
s.center(10,"*")
```

Output: 'Price: $ 356.09'

- ljust(...):

```
S.ljust(width[, fillchar]) -> str
```

Return S left-justified in a string of length "width". Padding is done using the specified fill character. If none is given, a space will be used as default.

Examples:

```
s = "Training"
s.ljust(12)
```

Output: 'Training   '

```
s.ljust(12,":")
```

Output: 'Training::::'

- rjust(...):

```
S.rjust(width[, fillchar]) -> str
```

Return S right-justified in a string of length width. Padding is done using the specified fill character. The default value is again a space.

Examples:

```python
s = "Programming"
s.rjust(15)
```

Output: `'    Programming'`

```python
s.rjust(15, "~")
```

Output: `'~~~~Programming'`

- zfill(...):

  S.zfill(width) -> str

Pad a string S with zeros on the left, to fill a field of the specified width. The string S is never truncated. This method can be easily emulated with rjust.

Examples:

```python
account_number = "43447879"
account_number.zfill(12)
# can be emulated with rjust:
```

Output: `'000043447879'`

```python
account_number.rjust(12,"0")
```

Output: `'000043447879'`

FORMATTED STRING LITERALS

Python 3.6 introduces formatted string literals. They are prefixed with an 'f'. The formatting syntax is similar to the format strings accepted by str.format(). Like the format string of format method, they contain replacement fields formed with curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol. It's easiest to understand by looking at the following examples:

```python
price = 11.23
f"Price in Euro: {price}"
```

Output: `'Price in Euro: 11.23'`

```python
f"Price in Swiss Franks: {price * 1.086}"
```

Output: `'Price in Swiss Franks: 12.195780000000001'`

```python
f"Price in Swiss Franks: {price * 1.086:5.2f}"
```

Output: `'Price in Swiss Franks: 12.20'`

```python
for article in ["bread", "butter", "tea"]:
    print(f"{article:>10}:")
```

```
     bread:
    butter:
       tea:
```

WORKING WITH DICTIONARIES

Introduction

In this chapter of our Python tutorial we assume that you are familiar with Python dictionaries and `while` loops, as we have discussed them in our chapters Dictionaries and Loops.

We included this chapter to provide the learners with additional exercises of both dictionaries and `while` loops. The focus lies on nested dictionaries, in our case dictionaries of dictionaries. From the experiences of my courses, I know that these often cause special difficulties in particular for beginners.

This chapter is also about coffee, tea and other hot drinks, whose consumption we manage using Python Dictionaries.

```python
kaffeeliste = {"Peter": 0,
               "Eva": 0,
               "Franka": 0}

while True:
    name = input("Name: ")
    if name == "":
        break
    kaffeeliste[name] += 1
    print(kaffeeliste[name])

print("kaffeeliste: ", kaffeeliste)
```

1

1

2

1

kaffeeliste: {'Peter': 1, 'Eva': 1, 'Franka': 2}

```python
kaffeeliste = {"Peter": 0,
               "Eva": 0,
               "Franka": 0}
teeliste = {"Peter": 0,
            "Eva": 0,
            "Franka": 0}

while True:
    name = input("Name: ")
    if name == "":
        break
    getränk = input("Getränk (kaffee/tee): ")
    if getränk.lower() == "kaffee":
        kaffeeliste[name] += 1
        print(kaffeeliste[name])
    elif getränk.lower() == "tee":
        teeliste[name] += 1
        print(teeliste[name])
```

```python
print("Kaffeeliste: ", kaffeeliste)
print("Teeliste: ", teeliste)
```

1

1

1

```
Kaffeeliste:  {'Peter': 1, 'Eva': 1, 'Franka': 0}
Teeliste:  {'Peter': 0, 'Eva': 0, 'Franka': 1}
```

```python
getränkeliste = {"Peter": {"Tee": 0,
                           "Kaffee": 0},
                 "Eva": {"Tee": 0,
                         "Kaffee": 0},
                 "Franka": {"Tee": 0,
                            "Kaffee": 0}}

while True:
    name = input("Name: ").capitalize()
    if name == "":
        break
    if name not in getränkeliste:    # gibt keinen Key "name"
        antwort = input("Sollen wir " + name + " in Liste aufnehmen? (j/n)")
        if antwort in ["j", "ja", "Ja", "y"]:
            getränkeliste[name] = {"Tee": 0, "Kaffee": 0}
        else:
            print("Dann gibt's nichts zu trinken für " + name + "!")
            continue
    drink = input("Getränk (Kaffee/Tee): ").capitalize()
    getränkeliste[name][drink] += 1
print(getränkeliste)
```

```python
getränke = ["Tee", "Kaffee", "Kakao", "Gemüsebrühe"]
namen = ["Peter", "Eva", "Sarah", "Eddie", "Swen"]

getränkekonsum = {}
for name in namen:
    getränkekonsum[name] = {}
    print(getränkekonsum)
    for getränk in getränke:
        getränkekonsum[name][getränk] = 0
```

```
{'Peter': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0},
'Eva': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0},
'Eva': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sar
ah': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0},
'Eva': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sar
ah': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eddi
e': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0},
'Eva': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sar
ah': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eddi
e': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Swe
n': {}}
```

```python
supermarket = {"milk": {"quantity": 20, "price": 1.19},
               "biscuits":  {"quantity": 32, "price": 1.45},
               "butter":   {"quantity": 20, "price": 2.29},
               "cheese":   {"quantity": 15, "price": 1.90},
               "bread":   {"quantity": 15, "price": 2.59},
               "cookies":  {"quantity": 20, "price": 4.99},
               "yogurt": {"quantity": 18, "price": 3.65},
               "apples":   {"quantity": 35, "price": 3.15},
               "oranges":   {"quantity": 40, "price": 0.99},
               "bananas": {"quantity": 23, "price": 1.29}}
```

```python
total_value = 0
for article, numbers in supermarket.items():
    quantity = numbers["quantity"]
    price = numbers["price"]
    product_price = quantity * price
    article = article + ':'
    print(f"{article:15s} {product_price:08.2f}")
    total_value += product_price
print("="*24)
print(f"Gesamtsumme:    {total_value:08.2f}")
```
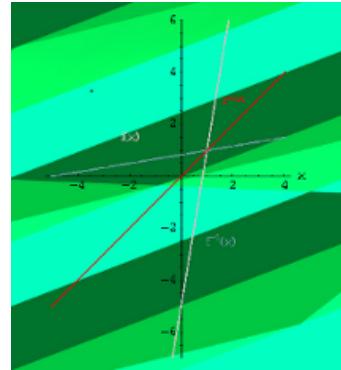
```
milk:           00023.80
biscuits:       00046.40
butter:         00045.80
cheese:         00028.50
bread:          00038.85
cookies:        00099.80
yogurt:         00065.70
apples:         00110.25
oranges:        00039.60
bananas:        00029.67
========================
Gesamtsumme:    00528.37
```

FUNCTIONS

SYNTAX

The concept of a function is one of the most important in mathematics. A common usage of functions in computer languages is to implement mathematical functions. Such a function is computing one or more results, which are entirely determined by the parameters passed to it.

This is mathematics, but we are talking about programming and Python. So what is a function in programming? In the most general sense, a function is a structuring element in programming languages to group a bunch of statements so they can be utilized in a program more than once. The only way to accomplish this without functions would be to reuse code by copying it and adapting it to different contexts, which would be a bad idea. Redundant code - repeating code in this case - should be avoided! Using functions usually enhances the comprehensibility and quality of a program. It also lowers the cost for development and maintenance of the software.

Functions are known under various names in programming languages, e.g. as subroutines, routines, procedures, methods, or subprograms.

MOTIVATING EXAMPLE OF FUNCTIONS

Let us look at the following code:

```python
print("Program starts")

print("Hi Peter")
print("Nice to see you again!")
print("Enjoy our video!")

# some lines of codes
the_answer = 42

print("Hi Sarah")
print("Nice to see you again!")
print("Enjoy our video!")

width, length = 3, 4
area = width * length

print("Hi Dominque")
print("Nice to see you again!")
```

```
print("Enjoy our video!")
```

```
Program starts
Hi Peter
Nice to see you again!
Enjoy our video!
Hi Sarah
Nice to see you again!
Enjoy our video!
Hi Dominque
Nice to see you again!
Enjoy our video!
```

Let us have a closer look at the code above:
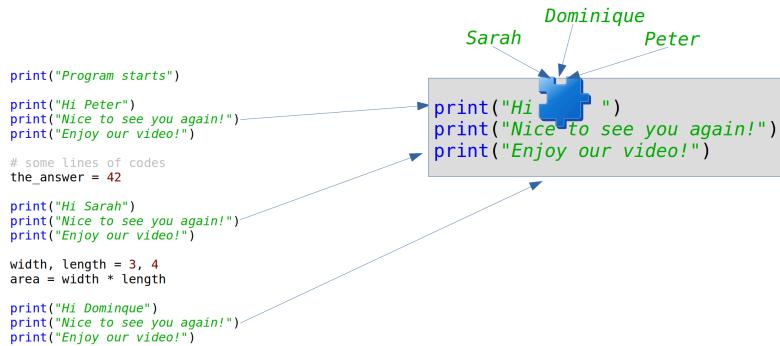
```
print("Program starts")

print("Hi Peter")
print("Nice to see you again!")
print("Enjoy our video!")

# some lines of codes
the_answer = 42

print("Hi Sarah")
print("Nice to see you again!")
print("Enjoy our video!")

just_some_code = "whatever"
another_var = "whatever you do"

print("Hi Dominque")
print("Nice to see you again!")
print("Enjoy our video!")
```

You can see in the code that we are greeting three persons. Every time we use three print calls which are nearly the same. Just the name is different. This is what we call redundant code. We are repeating the code the times. This shouldn't be the case. This is the point where functions can and should be used in Python.

We could use wildcards in the code instead of names. In the following diagram we use a piece of the puzzle. We only write the code once and replace the puzzle piece with the corresponding name:

Of course, this was no correct Python code. We show how to do this in Python in the following section.

FUNCTIONS IN PYTHON

The following code uses a function with the name `greet`. The previous puzzle piece is now a parameter with the name "name":

```python
def greet(name):
    print("Hi " + name)
    print("Nice to see you again!")
    print("Enjoy our video!")


print("Program starts")

greet("Peter")


# some lines of codes
the_answer = 42

greet("Sarah")

width, length = 3, 4
area = width * length


greet("Dominque")
```

```
Program starts
Hi Peter
Nice to see you again!
Enjoy our video!
Hi Sarah
Nice to see you again!
Enjoy our video!
Hi Dominque
Nice to see you again!
Enjoy our video!
```



```python
def greet(name):
    print("Hi " + name)
    print("Nice to see you again!")
    print("Enjoy our video!")

print("Program starts")
greet("Peter")
# some lines of codes
the_answer = 42
greet("Sarah")
width, length = 3, 4
area = width * length
greet("Dominque")
```
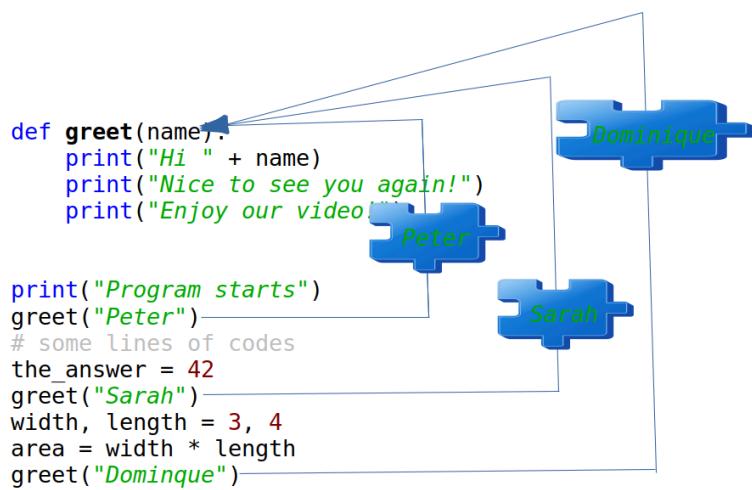
We used a function in the previous code. We saw that a function definition starts with the `def` keyword. The general syntax looks like this:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called. We demonstrate this in the following picture:

```
                                          def f(x, y):
                                              z = 2 * (x + y)
                                              return z

            print("Program starts!")
            a = 3
            res1 = f(a, 2+a)
            print("Result of function call:",
            res1)
            a = 4
            b = 7
            res2 = f(a, b)
            print("Result of function call:",
            res2)
```

The code from the picture can be seen in the following:

```python
def f(x, y):
    z = 2 * (x + y)
    return z


print("Program starts!")
a = 3
res1 = f(a, 2+a)
print("Result of function call:", res1)
a = 4
b = 7
res2 = f(a, b)
print("Result of function call:", res2)
```

```
Program starts!
Result of function call: 16
Result of function call: 22
```

We call the function twice in the program. The function has two parameters, which are called `x` and `y`. This means that the function `f` is expecting two values, or I should say "two objects". Firstly, we call this function with `f(a, 2+a)`. This means that `a` goes to `x` and the result of `2+a` (5) 'goes to' the variable `y`. The mechanism for assigning arguments to parameters is called **argument passing**. When we reach the `return` statement, the object referenced by `z` will be return, which means that it will be assigned to the variable `res1`. After leaving the function `f`, the variable `z` and the parameters `x` and `y` will be deleted automatically.
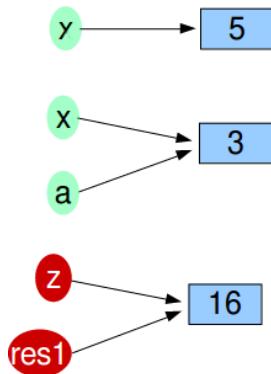
```python
                                           def f(x, y):
                                               z = 2 * (x + y)
                                               return z

        print("Program starts!")
        a = 3
        res1 = f(a, 2+a)
        print("Result of function call:",
        res1)
        a = 4
        b = 7
        res2 = f(a, b)
        print("Result of function call:",
        res2)
```

The references to the objects can be seen in the next diagram:

The next Python code block contains an example of a function without a return statement. We use the `pass` statement inside of this function. `pass` is a null operation. This means that when it is executed, nothing happens. It is useful as a placeholder in situations when a statement is required syntactically, but no code needs to be executed:

```python
    def doNothing():
        pass
```

A more useful function:

```python
def fahrenheit(T_in_celsius):
    """ returns the temperature in degrees Fahrenheit """
    return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
    print(t, ": ", fahrenheit(t))
```

```
22.6 :  72.68
25.8 :  78.44
27.3 :  81.14
29.8 :  85.64
```

""" returns the temperature in degrees Fahrenheit """ is the so-called docstring. It is used by the help function:

```
help(fahrenheit)
```

```
Help on function fahrenheit in module __main__:

fahrenheit(T_in_celsius)
    returns the temperature in degrees Fahrenheit
```

Our next example could be interesting for the calorie-conscious Python learners. We also had an exercise in the chapter Conditional Statements of our Python tutorial. We will create now a function version of the program.

The body mass index (BMI) is a value derived from the mass (w) and height (l) of a person. The BMI is defined as the body mass divided by the square of the body height, and is universally expressed

$$BMI = \frac{w}{l^2}$$

The weight is given in kg and the length in metres.

```
def BMI(weight, height):
    """ calculates the BMI where
        weight is in kg and height in metres"""
    return weight / height**2
```

We like to write a function to evaluate the bmi values according to the following table:

Category	BMI (kg/m²)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

The following shows code which is directly calculating the bmi and the evaluation of the bmi, but it is not using functions:

```
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

```
23.69576446280992
Normal (healthy weight)
```

Turn the previous code into proper functions and function calls.

```python
def BMI(weight, height):
    """ calculates the BMI where
        weight is in kg and height in metres"""
    return weight / height**2

def bmi_evaluate(bmi_value):
    if bmi_value < 15:
        result = "Very severely underweight"
    elif bmi_value < 16:
        result = "Severely underweight"
    elif bmi_value < 18.5:
        result = "Underweight"
    elif bmi_value < 25:
        result = "Normal (healthy weight)"
    elif bmi_value < 30:
        result = "Overweight"
    elif bmi_value < 35:
        result = "Obese Class I (Moderately obese)"
    elif bmi_value < 40:
        result = "Obese Class II (Severely obese)"
    else:
        result = "Obese Class III (Very severely obese)"
    return result
```

Let us check these functions:

```python
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

res = BMI(weight, height)
print(bmi_evaluate(res))
```

```
Normal (healthy weight)
```

DEFAULT ARGUMENTS IN PYTHON

When we define a Python function, we can set a default value to a parameter. If the function is called without the argument, this default value will be assigned to the parameter. This makes a parameter optional. To say it in other words: Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

We will demonstrate the operating principle of default parameters with a simple example. The following

function `hello` , - which isn't very useful, - greets a person. If no name is given, it will greet everybody:

```python
def hello(name="everybody"):
    """ Greets a person """
    result = "Hello " + name + "!")

hello("Peter")
hello()
```

```
Hello Peter!
Hello everybody!
```

THE DEFAULTS PITFALL

In the previous section we learned about default parameters. Default parameters are quite simple, but quite often programmers new to Python encounter a horrible and completely unexpected surprise. This surprise arises from the way Python treats the default arguments and the effects steming from mutable objects.

Mutable objects are those which can be changed after creation. In Python, dictionaries are examples of mutable objects. Passing mutable lists or dictionaries as default arguments to a function can have unforeseen effects. Programmer who use lists or dictionaries as default arguments to a function, expect the program to create a new list or dictionary every time that the function is called. However, this is not what actually happens. Default values will not be created when a function is called. Default values are created exactly once, when the function is defined, i.e. at compile-time.

Let us look at the following Python function "spammer" which is capable of creating a "bag" full of spam:

```python
def spammer(bag=[]):
    bag.append("spam")
    return bag
```

Calling this function once without an argument, returns the expected result:

```python
spammer()
```
Output: `['spam']`

The surprise shows when we call the function again without an argument:

```python
spammer()
```
Output: `['spam', 'spam']`

Most programmers will have expected the same result as in the first call, i.e. `['spam']`

To understand what is going on, you have to know what happens when the function is defined. The compiler creates an attribute `__defaults__`:

```python
def spammer(bag=[]):
    bag.append("spam")
    return bag

spammer.__defaults__
```

Output: `([],)`

Whenever we will call the function, the parameter `bag` will be assigned to the list object referenced by `spammer.__defaults__[0]`:

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

print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam', 'spam']
['spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam', 'spam']
spammer.__defaults__ (['spam', 'spam', 'spam', 'spam', 'spam'],)
```

Now, you know and understand what is going on, but you may ask yourself how to overcome this problem. The solution consists in using the immutable value `None` as the default. This way, the function can set bag dynamically (at run-time) to an empty list:

```python
def spammer(bag=None):
    if bag is None:
        bag = []
    bag.append("spam")
    return bag

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

print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam']
['spam']
['spam']
['spam']
spammer.__defaults__ (None,)
```

DOCSTRING

The first statement in the body of a function is usually a string statement called a Docstring, which can be accessed with the `function_name.__doc__`. For example:

```python
def hello(name="everybody"):
    """ Greets a person """
    print("Hello " + name + "!")

print("The docstring of the function hello: " + hello.__doc__)
```

```
The docstring of the function hello:  Greets a person
```

KEYWORD PARAMETERS

Using keyword parameters is an alternative way to make function calls. The definition of the function doesn't change. An example:

```python
def sumsub(a, b, c=0, d=0):
    return a - b + c - d

print(sumsub(12, 4))
print(sumsub(42, 15, d=10))
```

```
8
17
```

Keyword parameters can only be those, which are not used as positional arguments. We can see the benefit in the example. If we hadn't had keyword parameters, the second call to function would have needed all four arguments, even though the c argument needs just the default value:

```python
print(sumsub(42,15,0,10))
```

```
17
```

RETURN VALUES

In our previous examples, we used a return statement in the function sumsub but not in Hello. So, we can see

that it is not mandatory to have a return statement. But what will be returned, if we don't explicitly give a return statement. Let's see:

```
def no_return(x, y):
    c = x + y

res = no_return(4, 5)
print(res)
```

```
None
```

If we start this little script, *None* will be printed, i.e. the special value *None* will be returned by a return-less function. *None* will also be returned, if we have just a return in a function without an expression:

```
def empty_return(x, y):
    c = x + y
    return

res = empty_return(4, 5)
print(res)
```

```
None
```

Otherwise the value of the expression following return will be returned. In the next example 9 will be printed:

```
def return_sum(x, y):
    c = x + y
    return c

res = return_sum(4, 5)
print(res)
```

```
9
```

Let's summarize this behavior: Function bodies can contain one or more return statements. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value `None` is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value `None` will be returned.

RETURNING MULTIPLE VALUES

A function can return exactly one value, or we should better say one object. An object can be a numerical value, like an integer or a float. But it can also be e.g. a list or a dictionary. So, if we have to return, for example, 3 integer values, we can return a list or a tuple with these three integer values. That is, we can indirectly return multiple values. The following example, which is calculating the Fibonacci boundary for a positive number, returns a 2-tuple. The first element is the Largest Fibonacci Number smaller than x and the second component is the Smallest Fibonacci Number larger than x. The return value is immediately stored via unpacking into the variables lub and sup:

```python
def fib_interval(x):
    """ returns the largest fibonacci
    number smaller than x and the lowest
    fibonacci number higher than x"""
    if x < 0:
        return -1
    old, new = 0, 1
    while True:
        if new < x:
            old, new = new, old+new
        else:
            if new == x:
                new = old + new
            return (old, new)

while True:
    x = int(input("Your number: "))
    if x <= 0:
        break
    lub, sup = fib_interval(x)
    print("Largest Fibonacci Number smaller than x: " + str(lub))
    print("Smallest Fibonacci Number larger than x: " + str(sup))
```

```
Largest Fibonacci Number smaller than x: 5
Smallest Fibonacci Number larger than x: 8
```

LOCAL AND GLOBAL VARIABLES IN FUNCTIONS

Variable names are by default local to the function, in which they get defined.

```python
def f():
    print(s)       # free occurrence of s in f

s = "Python"
f()
```

Python

```python
def f():
    s = "Perl"      # now s is local in f
    print(s)

s = "Python"
f()
print(s)
```

```
Perl
Python
```

```python
def f():
    print(s)         # This means a free occurrence, contradiction
to bein local
    s = "Perl"       # This makes s local in f
    print(s)



s = "Python"
f()
print(s)
```

```
---------------------------------------------------------------------
UnboundLocalError                          Traceback (most recent call last)
<ipython-input-25-81b2fbbc4d42> in <module>
      6
      7 s = "Python"
----> 8 f()
      9 print(s)

<ipython-input-25-81b2fbbc4d42> in f()
      1 def f():
----> 2     print(s)
      3     s = "Perl"
      4     print(s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

If we execute the previous script, we get the error message: UnboundLocalError: local variable 's' referenced before assignment.

The variable s is ambigious in f(), i.e. in the first print in f() the global s could be used with the value "Python". After this we define a local variable s with the assignment s = "Perl".

```python
def f():
    global s
    print(s)
    s = "dog"
    print(s)
s = "cat"
f()
print(s)
```

```
cat
dog
dog
```

We made the variable s global inside of the script. Therefore anything we do to s inside of the function body of f is done to the global variable s outside of f.

```python
def f():
    global s
    print(s)
    s = "dog"       # globally changed
    print(s)

def g():
    s = "snake"     # local s
    print(s)

s = "cat"
f()
print(s)
g()
print(s)
```

```
cat
dog
dog
snake
dog
```

ARBITRARY NUMBER OF PARAMETERS

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple reference. This asterisk shouldn't be mistaken for the C syntax, where this notation is connected with pointers. Example:

```python
def arithmetic_mean(first, *values):
    """ This function calculates the arithmetic mean of a non-empt
y
        arbitrary number of numerical values """

    return (first + sum(values)) / (1 + len(values))

print(arithmetic_mean(45,32,89,78))
print(arithmetic_mean(8989.8,78787.78,3453,78778.73))
print(arithmetic_mean(45,32))
print(arithmetic_mean(45))
```

```
61.0
42502.3275
38.5
45.0
```

This is great, but we have still have one problem. You may have a list of numerical values. Like, for example,

```
x = [3, 5, 9]
```

You cannot call it with

```
arithmetic_mean(x)
```

because "arithmetic_mean" can't cope with a list. Calling it with

```
arithmetic_mean(x[0], x[1], x[2])
```
Output: 5.666666666666667

is cumbersome and above all impossible inside of a program, because list can be of arbitrary length.

The solution is easy: The star operator. We add a star in front of the x, when we call the function.

```
arithmetic_mean(*x)
```
Output: 5.666666666666667

This will "unpack" or singularize the list.

A practical example for `zip` and the star or asterisk operator: We have a list of 4, 2-tuple elements:

```
my_list = [('a', 232),
           ('b', 343),
           ('c', 543),
           ('d', 23)]
```

We want to turn this list into the following 2 element, 4-tuple list:

```
[('a', 'b', 'c', 'd'),
 (232, 343, 543, 23)]
```

This can be done by using the *-operator and the zip function in the following way:

```
list(zip(*my_list))
```
Output: `[('a', 'b', 'c', 'd'), (232, 343, 543, 23)]`

ARBITRARY NUMBER OF KEYWORD PARAMETERS

In the previous chapter we demonstrated how to pass an arbitrary number of positional parameters to a function. It is also possible to pass an arbitrary number of keyword parameters to a function as a dictionary. To this purpose, we have to use the double asterisk "**"

```
def f(**kwargs):
    print(kwargs)
```

```
f()
```

```
{}
```

```
f(de="German",en="English",fr="French")
```

```
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

One use case is the following:

```
def f(a, b, x, y):
    print(a, b, x, y)
d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

EXERCISES WITH FUNCTIONS

Rewrite the "dog age" exercise from chapter <u>Conditional Statements</u> as a function.
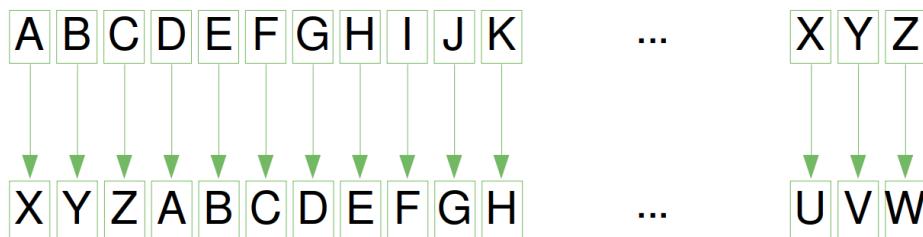
The rules are:

- A one-year-old dog is roughly equivalent to a 14-year-old human being
- A dog that is two years old corresponds in development to a 22 year old person.
- Each additional dog year is equivalent to five human years.

Write a function which takes a text and encrypts it with a Caesar cipher. This is one of the simplest and most commonly known encryption techniques. Each letter in the text is replaced by a letter some fixed number of positions further in the alphabet.

What about decrypting the coded text?

The Caesar cipher is a substitution cipher.

We can create another substitution cipher by permutating the alphet and map the letters to the corresponding permutated alphabet.

Write a function which takes a text and a dictionary to decrypt or encrypt the given text with a permutated alphabet.
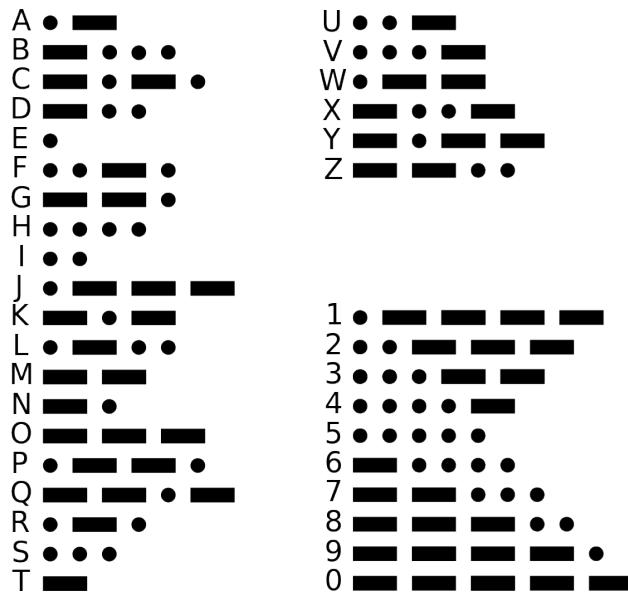
Write a function txt2morse, which translates a text to morse code, i.e. the function returns a string with the morse code.

Write another function morse2txt which translates a string in Morse code into a „normal" string.

The Morse character are separated by spaces. Words by three spaces.

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

Perhaps the first algorithm used for approximating $\sqrt{S}$ is known as the "Babylonian method", named after the Babylonians, or "Hero's method", named after the first-century Greek mathematician Hero of Alexandria who gave the first explicit description of the method.

If a number x_n is close to the square root of a then

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

will be a better approximation.

Write a program to calculate the square root of a number by using the Babylonian method.

Write a function which calculates the position of the n-th occurence of a string sub in another string s. If sub doesn't occur in s, -1 shall be returned.

Write a function `fuzzy_time` which expects a time string in the form hh: mm (e.g. "12:25", "04:56"). The function rounds up or down to a quarter of an hour. Examples:

fuzzy_time("12:58") ---> "13

SOLUTIONS

SOLUTION TO EXERCISE 1

```python
def dog_age2human_age(dog_age):
    """dog_age2human_age(dog_age)"""
    if dog_age == 1:
        human_age = 14
    elif dog_age == 2:
        human_age = 22
    else:
        human_age =  22 + (dog_age-2)*5
    return human_age

age = int(input("How old is your dog? "))
print(f"This corresponds to {dog_age2human_age(age)} human year
s!")
```

```
This corresponds to 37 human years!
```

```python
import string
abc = string.ascii_uppercase
def caesar(txt, n, coded=False):
    """ returns the coded or decoded text """
    result = ""
    for char in txt.upper():
        if char not in abc:
            result += char
```

```
        elif coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hello, here I am!", n)
print(x)
print(caesar(x, n, True))
```

```
EBIIL, EBOB F XJ!
HELLO, HERE I AM!
```

In the previous solution we only replace the letters. Every special character is left untouched. The following solution adds some special characters which will be also permutated. Special charcters not in abc will be lost in this solution!

```
import string
abc = string.ascii_uppercase + " .,-?!"
def caesar(txt, n, coded=False):
    """ returns the coded or decoded text """
    result = ""
    for char in txt.upper():
        if coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hello, here I am!", n)
print(x)
print(caesar(x, n, True))

x = caesar("abcdefghijkl", n)
print(x)
```

```
EBIILZXEBOBXFX-J,
HELLO, HERE I AM!
-?!ABCDEFGHI
```

We will present another way to do it in the following implementation. The advantage is that there will be no calculations like in the previous versions. We do only lookups in the decrypted list 'abc_cipher':

```python
import string

shift = 3
abc = string.ascii_uppercase + " .,-?!"
abc_cipher = abc[-shift:] + abc[:-shift]
print(abc_cipher)

def caesar (txt, shift):
    """Encodes the text "txt" to caesar by shifting it 'shift' pos
itions """
    new_txt=""
    for char in txt.upper():
        position = abc.find(char)
        new_txt +=abc_cipher[position]
    return new_txt

n = 3
x = caesar("Hello, here I am!", n)
print(x)

x = caesar("abcdefghijk", n)
print(x)
```

```
-?!ABCDEFGHIJKLMNOPQRSTUVWXYZ .,
EBIILZXEBOBXFX-J,
-?!ABCDEFGH
```

```python
import string
from random import sample

alphabet = string.ascii_letters
permutated_alphabet = sample(alphabet, len(alphabet))

encrypt_dict = dict(zip(alphabet, permutated_alphabet))
decrypt_dict = dict(zip(permutated_alphabet, alphabet))

def encrypt(text, edict):
    """ Every character of the text 'text'
    is mapped to the value of edict. Characters
    which are not keys of edict will not change"""
    res = ""
    for char in text:
        res = res + edict.get(char, char)
```

```
        return res

# Donald Trump: 5:19 PM, September 9 2014
txt = """Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional 'global warming' is worse."""

ctext = encrypt(txt, encrypt_dict)
print(ctext + "\n")
print(encrypt(ctext, decrypt_dict))
```

```
OQlerQGGk yDd xVd nDdyxdkx
xVDdyx Ql xVd Fz xo hoxV hyGe
yle noGedl dynGdk. gdeQy HGyQrk
EQHxQolyG 'nGohyG MyDrQln' Qk MoDkd.

Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional 'global warming' is worse.
```

Alternative solution:

```
import string
alphabet = string.ascii_lowercase + "äöüß .?!\n"

def caesar_code(alphabet, c, text, mode="encoding"):
    text = text.lower()
    coded_alphabet = alphabet[-c:] + alphabet[0:-c]
    if mode == "encoding":
        encoding_dict = dict(zip(alphabet, coded_alphabet))
    elif mode == "decoding":
        encoding_dict = dict(zip(coded_alphabet, alphabet))
    print(encoding_dict)
    result = ""
    for char in text:
        result += encoding_dict.get(char, "")
    return result

txt2 = caesar_code(alphabet, 5, txt)
caesar_code(alphabet, 5, txt2, mode="decoding")
```

```
{'a': ' ', 'b': '.', 'c': '?', 'd': '!', 'e': '\n', 'f': 'a',
'g': 'b', 'h': 'c', 'i': 'd', 'j': 'e', 'k': 'f', 'l': 'g', 'm':
'h', 'n': 'i', 'o': 'j', 'p': 'k', 'q': 'l', 'r': 'm', 's': 'n',
't': 'o', 'u': 'p', 'v': 'q', 'w': 'r', 'x': 's', 'y': 't', 'z':
'u', 'ä': 'v', 'ö': 'w', 'ü': 'x', 'ß': 'y', ' ': 'z', '.': 'ä',
'?': 'ö', '!': 'ü', '\n': 'ß'}
{' ': 'a', '.': 'b', '?': 'c', '!': 'd', '\n': 'e', 'a': 'f',
'b': 'g', 'c': 'h', 'd': 'i', 'e': 'j', 'f': 'k', 'g': 'l', 'h':
'm', 'i': 'n', 'j': 'o', 'k': 'p', 'l': 'q', 'm': 'r', 'n': 's',
'o': 't', 'p': 'u', 'q': 'v', 'r': 'w', 's': 'x', 't': 'y', 'u':
'z', 'v': 'ä', 'w': 'ö', 'x': 'ü', 'y': 'ß', 'z': ' ', 'ä': '.',
'ö': '?', 'ü': '!', 'ß': '\n'}
```

Output: `'windmills are the greatest \nthreat in the us to both bald \nand golden eagles. media claims \nfictional global warming is worse.'`

```python
latin2morse_dict = {'A':'.-', 'B':'-...', 'C':'-.-.', 'D':'-..',
                    'E':'.', 'F':'..-.', 'G':'--.','H':'....',
                    'I':'..', 'J':'.---', 'K':'-.-', 'L':'.-..',
                    'M':'--', 'N':'-.', 'O':'---', 'P':'.--.',
                    'Q':'--.-', 'R':'.-.', 'S':'...', 'T':'-',
                    'U':'..-', 'V':'...-', 'W':'.--', 'X':'-..-',
                    'Y':'-.--', 'Z':'--..', '1':'.----',
'2':'...--',
                    '3':'...--', '4':'....-', '5':'.....', '6':'-
....',
                    '7':'--...', '8':'---..', '9':'----.',
'0':'-----',
                    ',':'--..--', '.':'.-.-.-', '?':'..--..',
';':'-.-.-',
                    ':':'---...', '/':'-..-.', '-':'-....-',
'\'':'.----.',
                    '(':'-.--.-', ')':'-.--.-', '[':'-.--.-',
']':'-.--.-',
                    '{':'-.--.-', '}':'-.--.-', '_':'..--.-'}

# reversing the dictionary:
morse2latin_dict = dict(zip(latin2morse_dict.values(),
                           latin2morse_dict.keys()))

print(morse2latin_dict)
```

```
{'.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '..-
.': 'F', '--.': 'G', '....': 'H', '..': 'I', '.---': 'J', '-.-':
'K', '.-..': 'L', '--': 'M', '-.': 'N', '---': 'O', '.--.': 'P',
'--.-': 'Q', '.-.': 'R', '...': 'S', '-': 'T', '..-': 'U', '...-
': 'V', '.--': 'W', '-..-': 'X', '-.--': 'Y', '--..': 'Z', '.----
': '1', '...--': '3', '....-': '4', '.....': '5', '-....': '6',
'--...': '7', '---..': '8', '----.': '9', '-----': '0', '--..--':
',', '.-.-.-': '.', '..--..': '?', '-.-.-': ';', '---...': ':', '-
..-.': '/', '-....-': '-', '.----.': "'", '-.--.-': '}', '..--.-
': '_'}
```

```python
def txt2morse(txt, alphabet):
    morse_code = ""
    for char in txt.upper():
        if char == " ":
            morse_code += "   "
        else:
            morse_code += alphabet[char] + " "
    return morse_code

def morse2txt(txt, alphabet):
    res = ""
    mwords = txt.split("   ")
    for mword in mwords:
        for mchar in mword.split():
            res += alphabet[mchar]
        res += " "
    return res

mstring = txt2morse("So what?", latin2morse_dict)
print(mstring)
print(morse2txt(mstring, morse2latin_dict))
```

```
... ---   .-- .... .- - ..--..
SO WHAT?
```

```python
def heron(a, eps=0.000000001):
    """ Approximate the square root of a"""
    previous = 0
    new = 1
    while abs(new - previous) > eps:
        previous = new
        new = (previous + a/previous) / 2
```

```
    return new

print(heron(2))
print(heron(2, 0.001))

1.414213562373095
1.4142135623746899
```

```
def findnth(s, sub, n):
    num = 0
    start = -1
    while num < n:
        start = s.find(sub, start+1)
        if start == -1:
            break
        num += 1

    return start

s = "abc xyz abc jkjkjk abc lkjkjlkj abc jlj"
print(findnth(s,"abc", 3))
```

```
19
```

```
def vage_uhrzeit(zeit):
    stunden, minuten = zeit.split(":")
    minuten = int(minuten)
    stunden = int(stunden)
    minuten = minuten / 60    # Wandlung in Dezimalminuten
    # Werte in 0.0, 0.25, 0.5, 0.75:
    minuten = round(minuten * 4, 0) / 4
    if minuten == 0:
        print("Ungefähr " + str(stunden) + " Uhr!")
    elif minuten == 0.25:
        print("Ungefähr viertel nach " + str(stunden) + "!")
    else:
        if stunden == 12:
            stunden = 1
        else:
            stunden += 1
        if minuten == 0.50:
            print("Es ist ungefähr halb " + str(stunden) + "!")
        elif minuten == 0.75:
```

```
            print("Ungefähr eine viertel vor " + str(stunden) +
"!")
        elif minuten == 1.00:
            print("Ungefähr " + str(stunden) + " Uhr!")

stunde = "12"
for x in range(0, 60, 3):
    vage_uhrzeit(stunde +":" + f"{x:02d}")
```

```
Ungefähr 12 Uhr!
Ungefähr 12 Uhr!
Ungefähr 12 Uhr!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr 1 Uhr!
Ungefähr 1 Uhr!
```

RECURSIVE FUNCTIONS

DEFINITION

Recursion has something to do with infinity. I know recursion has something to do with infinity. I think I know recursion has something to do with infinity. He is sure I think I know recursion has something to do with infinity. We doubt he is sure I think I know ... We think that, you think , we convinced you now that, we can go on forever with this example of a recursion from natural language. Recursion is not only a fundamental feature of natural language, but of the human cognitive capacity. Our way of thinking is based on a recursive thinking processes. Even with a very simple grammar rule, like "An English sentence contains a subject and a predicate, and a predicate contains a verb, an object and a complement", we can demonstrate the infinite possibilities of the natural language. The cognitive scientist and linguist Stephen Pinker phrases it like this: "With a few thousand nouns that can fill the subject slot and a few thousand verbs that can fill the predicate slot, one already has several million ways to open a sentence. The possible combinations quickly multiply out to unimaginably large numbers. Indeed, the repertoire of sentences is theoretically infinite, because the rules of language use a trick called recursion. A recursive rule allows a phrase to contain an example of itself, as in She thinks that he thinks that they think that he knows and so on, ad infinitum. And if the number of sentences is infinite, the number of possible thoughts and intentions is infinite too, because virtually every sentence expresses a different thought or intention."[1]

We have to stop our short excursion to the use of recursion in natural language to come back to recursion in computer science and programs and finally to recursion in the programming language Python.

The adjective "recursive" originates from the Latin verb "recurrere", which means "to run back". And this is what a recursive definition or a recursive function does: It is "running back" or returning to itself. Most people who have done some mathematics, computer science or read a book about programming will have encountered the factorial, which is defined in mathematical terms as

```
n! = n * (n-1)!, if n > 1 and 0! = 1
```

It's used so often as an example for recursion because of its simplicity and clarity.

DEFINITION OF RECURSION

Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition: A recursive function has to fulfil an important condition to be used in a program: it has

to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can end up in an infinite loop, if the base case is not met in the calls.

Example:

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

Replacing the calculated values gives us the following expression

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

In other words, recursion in computer science is a method where the solution to a problem is based on solving smaller instances of the same problem.

RECURSIVE FUNCTIONS IN PYTHON

Now we come to implement the factorial in Python. It's as easy and elegant as the mathematical definition.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We can track how the function works by adding two print() functions to the previous function definition:

```python
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial("
,n-1, "): ",res)
        return res

print(factorial(5))
```

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120
120
```

Let's have a look at an iterative version of the factorial function.

```python
def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result


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

```
0 1
1 1
2 2
3 6
4 24
```

It is common practice to extend the factorial function for 0 as an argument. It makes sense to define `0!` to be `1`, because there is exactly one permutation of zero objects, i.e. if nothing is to permute, "everything" is left in place. Another reason is that the number of ways to choose n elements among a set of n is calculated as n! divided by the product of n! and 0!.

All we have to do to implement this is to change the condition of the if statement:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

FIBONACCI NUMBERS

Our treatise of recursion leads us now to another interesting case of recursion.

What do have sunflowers, the Golden ratio, fir tree cones, The Da Vinci Code, the song "Lateralus" by Tool, and the graphic on the right side in common? Right, the Fibonacci numbers. We are not introducing the Fibonacci numbers, because they are another useful example for recusive function. Trying to write a recursive function for this sequence of numbers, might lead to an inefficient program. In fact, so inefficient that it will not be useful. We introduce the Fibonacci numbers to show you the pitfalls of recursion.
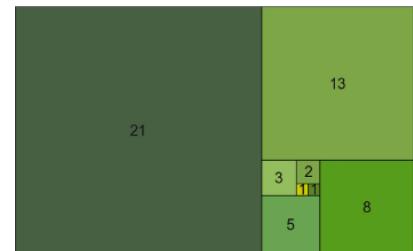
The Fibonacci numbers are the numbers of the following sequence of integer values:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The Fibonacci numbers are defined by:

$$F_n = F_{n-} + F_{n-2}$$

with $F_0 = 0$ and $F_1 = 1$

The Fibonacci sequence is named after the mathematician Leonardo of Pisa, who is better known as Fibonacci. In his book "Liber Abaci" (published in 1202) he introduced the sequence as an exercise dealing with bunnies. His sequence of the Fibonacci numbers begins with F1 = 1, while in modern mathematics the sequence starts with F0 = 0. But this has no effect on the other members of the sequence.

The Fibonacci numbers are the result of an artificial rabbit population, satisfying the following conditions:

- a newly born pair of rabbits, one male, one female, build the initial population
- these rabbits are able to mate at the age of one month so that at the end of its second month a female can bring forth another pair of rabbits
- these rabbits are immortal
- a mating pair always produces one new pair (one male, one female) every month from the second month onwards

The Fibonacci numbers are the numbers of rabbit pairs after `n` months, i.e. after 10 months we will have $F_1 0$ rabits.

The Fibonacci numbers are easy to write as a Python function. It's more or less a one to one mapping from the mathematical definition:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

An iterative solution is also easy to write, though the recursive solution looks more like the definition:

```python
def fibi(n):
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        old, new = new, old + new
    return new
```

We will write a module with the name `fibonacci` containing both the funciton `fib` and `fibi`. To do this you have to copy the following code into a file with the name fibonacci0.py:

```python
""" A module containing both a recursive and an iterative implemen
tation of the Fibonacci function.
The purpose of this module consists in showing the inefficiency o
f a purely recursive implementation of Fibonacci! """

def fib(n):
    """ recursive version of the Fibonacci function """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fibi(n):
    """ iterative version of the Fibonacci function """
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        old, new = new, old + new
    return new
```

```
Overwriting fibonacci0.py
```

If you check the functions fib() and fibi(), you will find out that the iterative version fibi() is a lot faster than the recursive version fib(). To get an idea of how much this "a lot faster" can be, we have written a script, which uses the timeit module, to measure the calls. To do this, we save the function definitions for fib and fibi in a file fibonacci.py, which we can import in the program (fibonacci_runit.py) below:

```
from timeit import Timer

t1 = Timer("fib(10)","from fibonacci import fib")

for i in range(1, 20):
    cmd = "fib(" + str(i) + ")"
    t1 = Timer(cmd, "from fibonacci import fib")
    time1 = t1.timeit(3)
    cmd = "fibi(" + str(i) + ")"
    t2 = Timer(cmd, "from fibonacci import fibi")
    time2 = t2.timeit(3)
    print(f"n={i:2d}, fib: {time1:8.6f}, fibi:  {time2:7.6f}, time
1/time2: {time1/time2:10.2f}")
```
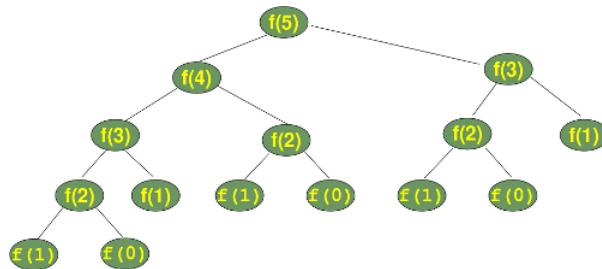
```
n= 1,  fib: 0.000001,  fibi:  0.000002, time1/time2:        0.50
n= 2,  fib: 0.000002,  fibi:  0.000002, time1/time2:        0.85
n= 3,  fib: 0.000003,  fibi:  0.000002, time1/time2:        1.11
n= 4,  fib: 0.000004,  fibi:  0.000003, time1/time2:        1.45
n= 5,  fib: 0.000011,  fibi:  0.000003, time1/time2:        3.35
n= 6,  fib: 0.000008,  fibi:  0.000003, time1/time2:        2.98
n= 7,  fib: 0.000012,  fibi:  0.000002, time1/time2:        5.05
n= 8,  fib: 0.000018,  fibi:  0.000002, time1/time2:        7.98
n= 9,  fib: 0.000029,  fibi:  0.000002, time1/time2:       12.15
n=10,  fib: 0.000046,  fibi:  0.000003, time1/time2:       18.18
n=11,  fib: 0.000075,  fibi:  0.000003, time1/time2:       28.42
n=12,  fib: 0.000153,  fibi:  0.000003, time1/time2:       51.99
n=13,  fib: 0.000194,  fibi:  0.000003, time1/time2:       72.11
n=14,  fib: 0.000323,  fibi:  0.000003, time1/time2:      109.50
n=15,  fib: 0.001015,  fibi:  0.000008, time1/time2:      134.65
n=16,  fib: 0.002165,  fibi:  0.000008, time1/time2:      261.11
n=17,  fib: 0.003795,  fibi:  0.000019, time1/time2:      198.87
n=18,  fib: 0.006021,  fibi:  0.000010, time1/time2:      574.59
n=19,  fib: 0.008807,  fibi:  0.000011, time1/time2:      785.54
```

`time1` is the time in seconds it takes for 3 calls to `fib(n)` and `time2` respectively the time for `fibi(n)`.

What's wrong with our recursive implementation?

Let's have a look at the calculation tree, i.e. the order in which the functions are called. fib() is substituted by f().

We can see that the subtree f(2) appears 3 times and the subtree for the calculation of f(3) two times. If you imagine extending this tree for f(6), you will understand that f(4) will be called two times, f(3) three times and so on. This means, our recursion doesn't remember previously calculated values.

We can implement a "memory" for our recursive version by using a dictionary to save the previously calculated values. We call this version `fibm`:

```python
memo = {0:0, 1:1}
def fibm(n):
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```

Before we can do some timing on the new version, we add it to our `fibonacci` module:

```python
""" A module containing both a recursive and an iterative implemen
tation of the Fibonacci function.
The purpose of this module consists in showing the inefficiency o
f a purely recursive implementation of Fibonacci! """

def fib(n):
    """ recursive version of the Fibonacci function """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fibi(n):
    """ iterative version of the Fibonacci function """
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        old, new = new, old + new
```

```
        return new

memo = {0:0, 1:1}
def fibm(n):
    """ recursive Fibonacci function which memoizes previously
    calculated values with the help of a dictionary memo"""
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```

Overwriting fibonacci.py

We time it again to compare it with fibi():

```
from timeit import Timer
from fibonacci import fib

t1 = Timer("fib(10)","from fibonacci import fib")

for i in range(1, 20):
    s = "fibm(" + str(i) + ")"
    t1 = Timer(s,"from fibonacci import fibm")
    time1 = t1.timeit(3)
    s = "fibi(" + str(i) + ")"
    t2 = Timer(s,"from fibonacci import fibi")
    time2 = t2.timeit(3)
    print(f"n={i:2d}, fibm: {time1:8.6f}, fibi:  {time2:7.6f}, tim
e1/time2: {time1/time2:10.2f}")
```

```
n= 1, fibm: 0.000002, fibi:  0.000002, time1/time2:    0.68
n= 2, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.63
n= 3, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.51
n= 4, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.46
n= 5, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.48
n= 6, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.41
n= 7, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.41
n= 8, fibm: 0.000001, fibi:  0.000011, time1/time2:    0.10
n= 9, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.40
n=10, fibm: 0.000001, fibi:  0.000002, time1/time2:    0.36
n=11, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.38
n=12, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.35
n=13, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.44
n=14, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.36
n=15, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.35
n=16, fibm: 0.000001, fibi:  0.000004, time1/time2:    0.34
n=17, fibm: 0.000001, fibi:  0.000003, time1/time2:    0.33
n=18, fibm: 0.000001, fibi:  0.000004, time1/time2:    0.29
n=19, fibm: 0.000001, fibi:  0.000004, time1/time2:    0.27
```

We can see that it is even faster than the iterative version. Of course, the larger the arguments, the greater the benefit of our memorization:

We can also define a recursive algorithm for our Fibonacci function by using a class with callabe instances, i.e. by using the special method **call**. This way, we will be able to hide the dictionary in an elegant way. We used a general approach which allows as to define also functions similar to Fibonacci, like the Lucas function. This means that we can create Fibonacci-like number sequences by instantiating instances of this class. Basically, the building principle, i.e. the sum of the previous two numbers, will be the same. They will differ by the two initial values:

```python
class FibonacciLike:

    def __init__(self, i1=0, i2=1):
        self.memo = {0:i1, 1:i2}

    def __call__(self, n):
        if n not in self.memo:
            self.memo[n] = self.__call__(n-1) + self.__cal
l__(n-2)
        return self.memo[n]


# We create a callable to create the Fibonacci numbers:
fib = FibonacciLike()
```

```python
# This will create a callable to create the Lucas number series:
lucas = FibonacciLike(2, 1)

for i in range(1, 16):
    print(i, fib(i), lucas(i))
```

```
1 1 1
2 1 3
3 2 4
4 3 7
5 5 11
6 8 18
7 13 29
8 21 47
9 34 76
10 55 123
11 89 199
12 144 322
13 233 521
14 377 843
15 610 1364
```

The Lucas numbers or Lucas series are an integer sequence named after the mathematician François Édouard Anatole Lucas (1842–91), who studied both that sequence and the closely related Fibonacci numbers. The Lucas numbers have the same creation rule as the Fibonacci number, i.e. the sum of the two previous numbers, but the values for 0 and 1 are different.

GENERALIZED FIBONACCI SEQUENCE

We will demonstrate now that our approach is also suitable to calculate arbitrary generalized Fibonacci sequences. The Fibonacci sequence depends on the two preceding values. We generalize this concept now by defining a k-Fib sequence in the following way

Given a fixed natural number `k` and `k >= 2`. Also given `k` initial values $i_0, i_1, \ldots i_{k-1}$

satisfying $F_k(0) = i_0, \ldots F_k(k-1) = i_{k-1}$

The function also needs k cofficients $c_0, c_1, \ldots c_{k-1}$

$F_k(n)$ is defined as

$$F_k(n) = a_0 \cdot F_k(n-1) + a_1 \cdot F_k(n-2) \ldots a_{k-1} \cdot F_k(n-k)$$

with $k <= n$

```
class kFibonacci:

    def __init__(self, k, initials, coefficients):
        self.memo = dict(zip(range(k), initials))
        self.coeffs = coefficients
        self.k = k

    def __call__(self, n):
        k = self.k
        if n not in self.memo:
            result = 0
            for coeff, i in zip(self.coeffs, range(1, k+1)):
                result += coeff * self.__call__(n-i)
            self.memo[n] = result
        return self.memo[n]

fib = kFibonacci(2, (0, 1), (1, 1))
lucas = kFibonacci(2, (2, 1), (1, 1))

for i in range(1, 16):
    print(i, fib(i), lucas(i))
```

```
1 1 1
2 1 3
3 2 4
4 3 7
5 5 11
6 8 18
7 13 29
8 21 47
9 34 76
10 55 123
11 89 199
12 144 322
13 233 521
14 377 843
15 610 1364
```

The sequence of Pell numbers start with 1, 2, 5, 12, 29, ...

The function is

$P(n) = 2 \cdot P(n - 1) + P(n - 2)$ with $P(0) = 1$ and $P(1) = 1$

It is easy to formulate this sequence with our `kFibonacci` class:

```
P = kFibonacci(2, (1, 2), (2, 1))
for i in range(10):
    print(i, P(i))
```

```
0 1
1 2
2 5
3 12
4 29
5 70
6 169
7 408
8 985
9 2378
```

We can create another interesting series by adding the sum of previous two Pell numbers between two Pell numbers P(i) and P(i+1). We get:

$$P(0), P(1), P(0) + P(1), P(2), P(1) + P(2), P(3), P(2) + P(3), \ldots$$

corresponding to: $1, 2, 3, 5, 7, 12, 17, 29, \ldots$

```
def nP(n):
    if n < 2:
        return P(n)
    else:
        i = n // 2 + 1
        if n % 2:   # n is odd
            return P(i)
        else:
            return P(i-1) + P(i-2)

for i in range(20):
    print(nP(i), end=", ")
```

```
1, 2, 3, 5, 7, 12, 17, 29, 41, 70, 99, 169, 239, 408, 577, 985, 13
93, 2378, 3363, 5741,
```

If you have a close look at the numbers, you can see that there is another rule in this sequence. We can define nP as

$nP(n) = 2 \cdot nP(n - 2) + nP(n - 4)$ with $n > 3$ and the start values $1, 2, 3, 5$

This is another easy application for our `kFibonacci` class:

```
nP = kFibonacci(4, (1, 2, 3, 5), (0, 2, 0, 1))
```

```
for i in range(20):
    print(nP(i), end=", ")
```

```
1, 2, 3, 5, 7, 12, 17, 29, 41, 70, 99, 169, 239, 408, 577, 985, 13
93, 2378, 3363, 5741,
```

An interesting mathematical fact: For every odd n the quotient of $P(n)$ by $P(n-1)$ is an approximation of $\sqrt{2}$.

If `n` is even, the quotient is an approximation of $1 + \dfrac{1}{\sqrt{2}}$

```
sqrt2 = 2 ** 0.5
print("Square root of 2: ", sqrt2)
print("Square root of 3: ", 1 + 1 / sqrt2)
for i in range(1, 20):
    print(nP(i) / nP(i-1), end=", ")
```

```
Square root of 2:  1.4142135623730951
Square root of 3:  1.7071067811865475
2.0, 1.5, 1.6666666666666667, 1.4, 1.7142857142857142, 1.416666666
6666667, 1.7058823529411764, 1.4137931034482758, 1.707317073170731
7, 1.4142857142857144, 1.707070707070707, 1.4142011834319526, 1.70
7112970711297, 1.4142156862745099, 1.707105719237435, 1.4142131979
695431, 1.7071069633883704, 1.4142136248948696, 1.707106749925661
6,
```

MORE ABOUT RECURSION IN PYTHON

If you want to learn more on recursion, we suggest that you try to solve the following exercises. Please do not peer at the solutions, before you have given your best. If you have thought about a task for a while and you are still not capable of solving the exercise, you may consult our sample solutions.

In our section "Advanced Topics" of our tutorial we have a comprehensive treatment of the game or puzzle "Towers of Hanoi". Of course, we solve it with a function using a recursive function. The "Hanoi problem" is special, because a recursive solution almost forces itself on the programmer, while the iterative solution of the game is hard to find and to grasp.

EXERCISES

Think of a recursive version of the function f(n) = 3 * n, i.e. the multiples

Write a recursive Python function that returns the sum of the first n integers. (Hint: The function will be similiar to the factorial function!)

Write a function which implements the Pascal's triangle:

```
                1

             1      1

          1      2      1

       1      3      3      1

    1      4      6      4      1

 1      5     10     10      5      1
```


The Fibonacci numbers are hidden inside of Pascal's triangle. If you sum up the coloured numbers of the following triangle, you will get the 7th Fibonacci number:

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10   5   1
1   6   15  20  15   6   1
```

Write a recursive program to calculate the Fibonacci numbers, using Pascal's triangle.

Implement a recursive function in Python for the sieve of Eratosthenes.

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a specified integer. It was created by the ancient Greek mathematician Eratosthenes. The algorithm to find all the prime numbers less than or equal to a given integer n:

1. Create a list of integers from two to n: 2, 3, 4, ..., n
2. Start with a counter i set to 2, i.e. the first prime number
3. Starting from i+i, count up by i and remove those numbers from the list, i.e. 2*i, 3*i, 4*i, etc..
4. Find the first number of the list following i. This is the next prime number.
5. Set i to the number found in the previous step
6. Repeat steps 3 and 4 until i is greater than n. (As an improvement: It's enough to go to the square root of n)
7. All the numbers, which are still in the list, are prime numbers

You can easily see that we would be inefficient, if we strictly used this algorithm, e.g. we will try to remove the multiples of 4, although they have been already removed by the multiples of 2. So it's enough to produce the multiples of all the prime numbers up to the square root of n. We can recursively create these sets.

Write a recursive function fib_indexfib(), which returns the index of a number in the Fibonacci sequence, if the number is an element of this sequence, and returns -1 if the number is not contained in it, i.e.

$fib(fib\_index(n)) == n$

The sum of the squares of two consecutive Fibonacci numbers is also a Fibonacci number, e.g. 2 and 3 are elements of the Fibonacci sequence and $2^2 + 3^3 = 13$ corresponds to Fib(7).Use the previous function to find the position of the sum of the squares of two consecutive numbers in the Fibonacci sequence.

Mathematical explanation: Let a and b be two successive Fibonacci numbers with a prior to b. The Fibonacci sequence starting with the number "a" looks like this:

```
0                    a
1                    b
2                a + b
3          a + 2b
4        2a + 3b
5            3a + 5b
6            5a + 8b
```

We can see that the Fibonacci numbers appear as factors for a and b. The n-th element in this sequence can be calculated with the following formula:

$$F(n) = Fib(n - 1) \cdot a + Fib(n) \cdot b$$

From this we can conclude that for a natural number n, n>1, the following holds true:

$$Fib(2 \cdot n + 1) = Fib(n)^2 + Fib(n + 1)^2$$

The **tribonacci numbers** are like the Fibonacci numbers, but instead of starting with two predetermined terms, the sequence starts with three predetermined terms and each term afterwards is the sum of the preceding three terms. The first few tribonacci numbers are:

0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012, ...

The **tetranacci numbers** start with four predetermined terms, each term afterwards being the sum of the preceding four terms. The first few tetranacci numbers are:

0, 0, 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490, 2872, 5536, 10671, 20569, 39648, ...

This continues in the same way for pentanacci, hexanacci, heptanacci, octanacci, and so on.

Write a function for the tribonacci and tetranacci numbers.

What if somebody wants to check the parameters in a recursive function? For example the factorial function. Okay, please write a recursive version of factorial, which checks the parameters. Maybe you did it in a perfect way, but maybe not. Can you improve your version? Get rid of unnecessary tests?

SOLUTIONS

SOLUTION TO EXERCISE 1

Mathematically, we can write it like this:

$f(1) = 3,$

$f(n + 1) = f(n) + 3$

A Python function can be written like this:

```python
def mult3(n):
    if n == 1:
        return 3
    else:
```

```
        return mult3(n-1) + 3

for i in range(1,10):
        print(mult3(i))
```

```
3
6
9
12
15
18
21
24
27
```

```
def sum_n(n):
    if n== 0:
        return 0
    else:
        return n + sum_n(n-1)
```

```
def pascal(n):
    if n == 1:
        return [1]
    else:
        line = [1]
        previous_line = pascal(n-1)
        for i in range(len(previous_line)-1):
            line.append(previous_line[i] + previous_line[i+1])
        line += [1]
    return line

print(pascal(6))
```

```
[1, 5, 10, 10, 5, 1]
```

Alternatively, we can write a function using list comprehension:

```
def pascal(n):
    if n == 1:
        return [1]
```

```python
    else:
        p_line = pascal(n-1)
        line = [ p_line[i]+p_line[i+1] for i in range(len(p_lin
e)-1)]
        line.insert(0,1)
        line.append(1)
    return line

print(pascal(6))
```

```
[1, 5, 10, 10, 5, 1]
```

Producing the Fibonacci numbers out of Pascal's triangle:

```python
def fib_pascal(n,fib_pos):
    if n == 1:
        line = [1]
        fib_sum = 1 if fib_pos == 0 else 0
    else:
        line = [1]
        (previous_line, fib_sum) = fib_pascal(n-1, fib_pos+1)
        for i in range(len(previous_line)-1):
            line.append(previous_line[i] + previous_line[i+1])
        line += [1]
        if fib_pos < len(line):
            fib_sum += line[fib_pos]
    return (line, fib_sum)

def fib(n):
    return fib_pascal(n,0)[1]

# and now printing out the first ten Fibonacci numbers:
for i in range(1,10):
    print(fib(i))
```

```
1
1
2
3
5
8
13
21
34
```

The following program implements the sieve of Eratosthenes according to the rules of the task in an iterative way. It will print out the first 100 prime numbers.

```python
from math import sqrt
def sieve(n):
    # returns all primes between 2 and n
    primes = list(range(2,n+1))
    max = sqrt(n)
    num = 2
    while num < max:
        i = num
        while i <= n:
            i += num
            if i in primes:
                primes.remove(i)
        for j in primes:
            if j > num:
                num = j
                break
    return primes

print(sieve(100))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
1, 67, 71, 73, 79, 83, 89, 97]
```

But this chapter of our tutorial is about recursion and recursive functions, and we have demanded a recursive function to calculate the prime numbers. To understand the following solution, you may confer our chapter about List Comprehension:

```python
from math import sqrt
```

```
def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return []
    else:
        p = primes(int(sqrt(n)))
        no_p = [j for i in p for j in range(i*2, n + 1, i)]
        p = [x for x in range(2, n + 1) if x not in no_p]
        return p


print(primes(100))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
1, 67, 71, 73, 79, 83, 89, 97]
```

```
memo = {0:0, 1:1}
def fib(n):
    if not n in memo:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

def fib_index(*x):
    """ finds the natural number i with fib(i) == n """
    if len(x) == 1:
        # started by user
        # find index starting from 0
        return fib_index(x[0], 0)
    else:
        n = fib(x[1])
        m = x[0]
        if  n > m:
            return -1
        elif n == m:
            return x[1]
        else:
            return fib_index(m, x[1]+1)


# code from the previous example with the functions fib() and fin
d_index()
```

```python
print(" index of a |    a |      b | sum of squares | index
")
print("======================================================")
for i in range(15):
    square = fib(i)**2 + fib(i+1)**2
    print(f"{i:12d}|{fib(i):6d}|{fib(i+1):9d}|{square:14d}|{fib_in
dex(square):5d}")
```

```
 index of a |    a |      b | sum of squares | index
======================================================
          0|     0|      1|             1|    1
          1|     1|      1|             2|    3
          2|     1|      2|             5|    5
          3|     2|      3|            13|    7
          4|     3|      5|            34|    9
          5|     5|      8|            89|   11
          6|     8|     13|           233|   13
          7|    13|     21|           610|   15
          8|    21|     34|          1597|   17
          9|    34|     55|          4181|   19
         10|    55|     89|         10946|   21
         11|    89|    144|         28657|   23
         12|   144|    233|         75025|   25
         13|   233|    377|        196418|   27
         14|   377|    610|        514229|   29
```

We will use the class `kFibonacci` which we have defined in this chapter.

```python
tribonacci = kFibonacci(3, (0, 0, 1), (1, 1, 1))

for i in range(20):
    print(tribonacci(i), end=", ")
```

```
0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 313
6, 5768, 10609, 19513,
```

```python
tetranacci = kFibonacci(4, (0, 0, 0, 1), (1, 1, 1, 1))

for i in range(20):
    print(tetranacci(i), end=", ")
```

```
0, 0, 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490, 287
2, 5536, 10671, 20569,
```

We can easily create them like this:

```python
prefixes = ['tribo', 'tetra', 'pentra', 'hexa', 'hepta', 'octa']
for k, prefix in zip(range(len(prefixes)), prefixes):
    cmd = prefix + "nacci = kFibonacci("
    cmd += str(k+3) + ", (" + "0, " * (k+2) + "1), "
    cmd += "(" + "1, " * (k+2) + "1))"
    print(cmd)
    exec(cmd)

print("\nTesting the octanacci function: ")
for i in range(20):
    print(octanacci(i), end=", ")
```

```
tribonacci = kFibonacci(3, (0, 0, 1), (1, 1, 1))
tetranacci = kFibonacci(4, (0, 0, 0, 1), (1, 1, 1, 1))
pentranacci = kFibonacci(5, (0, 0, 0, 0, 1), (1, 1, 1, 1, 1))
hexanacci = kFibonacci(6, (0, 0, 0, 0, 0, 1), (1, 1, 1, 1, 1, 1))
heptanacci = kFibonacci(7, (0, 0, 0, 0, 0, 0, 1), (1, 1, 1, 1, 1,
1, 1))
octanacci = kFibonacci(8, (0, 0, 0, 0, 0, 0, 0, 1), (1, 1, 1, 1,
1, 1, 1, 1))

Testing the octanacci function:
0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 4, 8, 16, 32, 64, 128, 255, 509, 101
6, 2028,
```

```python
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    if type(n) == int and n >=0:
        if n == 0:
            return 1
        else:
            return n * factorial(n-1)

    else:
        raise TypeError("n has to be a positive integer or zero")
```

What do you think about this solution? You call `factorial(10)` for example. The funciton will check, if `10` is a positive integer. The function will recursively call `factorial(9)`. In this call the function will check again, if `9` is an integer. What else can it be? The first call had already checked `10` and all we did

was subtracting `1` from `10` .

We can overcome this problem by defining an inner function:

```python
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    def inner_factorial(n):
        if n == 0:
            return 1
        else:
            return n * inner_factorial(n-1)
    if type(n) == int and n >=0:
        return inner_factorial(n)
    else:
        raise TypeError("n should be a positve int or 0")

print(factorial(10))
```

```
3628800
```

The test will only be performed on the first call with `10` !

*Stephen Pinker, The Blank Slate, 2002

PARAMETERS AND ARGUMENTS
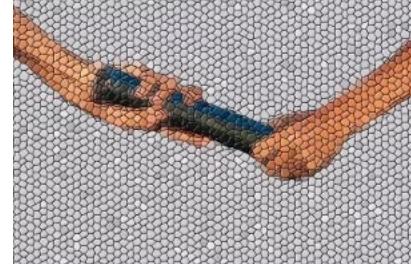
A function or procedure usually needs some information about the environment, in which it has been called. The interface between the environment, from which the function has been called, and the function, i.e. the function body, consists of special variables, which are called parameters. By using these parameters, it's possible to use all kind of objects from "outside" inside of a function. The syntax for how parameters are declared and the semantics for how the arguments are passed to the parameters of the function or procedure depends on the programming language.

Very often the terms parameter and argument are used synonymously, but there is a clear difference. Parameters are inside functions or procedures, while arguments are used in procedure calls, i.e. the values passed to the function at run-time.

"CALL BY VALUE" AND "CALL BY NAME"

The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs between programming languages. The most common evaluation strategies are "call by value" and "call by reference":

- Call by Value The most common strategy is the call-by-value evaluation, sometimes also called pass-by-value. This strategy is used in C and C++, for example. In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will stay unchanged when the function returns.

- Call by Reference In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed. By using Call by Reference we save both computation time and memory space, because arguments do not need to be copied. On the other hand this harbours the disadvantage that variables can be "accidentally" changed in a function call. So, special care has to be taken to "protect" the values, which shouldn't be changed. Many programming languages support call-by-reference, like C or C++, but Perl uses it as default.

In ALGOL 60 and COBOL there has been a different concept called call-by-name, which isn't used anymore in modern languages.

AND WHAT ABOUT PYTHON?
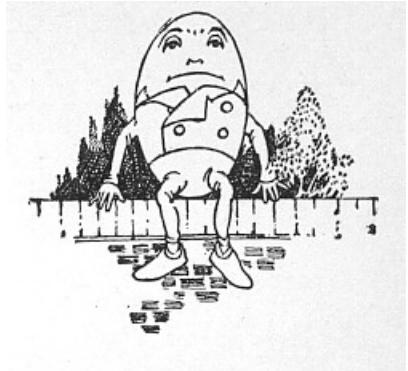
There are some books which call the strategy of Python call-by-value, and some call it call-by-reference. You may ask yourself, what is right.

Humpty Dumpty supplies the explanation:

--- "When I use a word," Humpty Dumpty said, in a rather a scornful tone, "it means just what I choose it to mean - neither more nor less."

--- "The question is," said Alice, "whether you can make words mean so many different things."

--- "The question is," said Humpty Dumpty, "which is to be master - that's all."
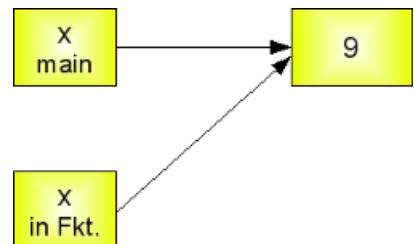
Lewis Carroll, Through the Looking-Glass

To come back to our initial question what evaluation strategy is used in Python: the authors who call the mechanism call-by-value and those who call it call-by-reference are stretching the definitions until they fit.

Correctly speaking, Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place within the function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.
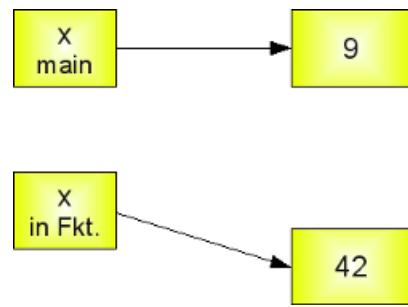
First, let's have a look at the integer variables below. The parameter inside the function remains a reference to the argument's variable, as long as the parameter is not changed. As soon as a new value is assigned to it, Python creates a separate local variable. The caller's variable will not be changed this way:

```python
def ref_demo(x):
    print("x=",x," id=",id(x))
    x=42
    print("x=",x," id=",id(x))
```

In the example above, we used the id() function, which takes an object as a parameter. id(obj) returns the "identity" of the object "obj". This identity, the return value of the function, is an integer which is unique and constant for this object during its lifetime. Two different objects with non-overlapping lifetimes may have the same id() value.

If you call the function ref_demo() of the previous example - like we do in the green block further below - we can check what happens to x with the id() function: We can see that in the main scope, x has the identity 140709692940944. In the first print statement of the ref_demo() function, the x from the main scope is used, because we can see that we get the same identity. After we assigned the value 42 to x, x gets a new identity 140709692942000, i.e. a separate memory location from the global x. So, when we are back in the main scope x has still the original value 9 and the id 140709692940944.

In other words, Python initially behaves like call-by-reference, but as soon as we change the value of such a variable, i.e. as soon as we assign a new object to it, Python "switches" to call-by-value. That is, a local variable x will be created and the value of the global variable x will be copied into it.

```
x = 9
id(x)
```

Output: 140709692940944

```
ref_demo(x)
```

```
x= 9   id= 140709692940944
x= 42   id= 140709692942000
```

```
id(x)
```

Output: 140709692940944

SIDE EFFECTS

A function is said to have a side effect, if, in addition to producing a return value, it modifies the caller's environment in other ways. For example, a function might modify a global or static variable, modify one of its arguments, raise an exception, write data to a display or file etc.

There are situations, in which these side effects are intended, i.e. they are part of the function's specification. But in other cases, they are not wanted , they are hidden side effects. In this chapter, we are only interested in the side effects that change one or more global variables, which have been passed as arguments to a function. Let's assume, we are passing a list to a function. We expect the function not to change this list. First, let's have a look at a function which has no side effects. As a new list is assigned to the parameter list in func1(), a new memory location is created for list and list becomes a local variable.

```
def no_side_effects(cities):
```

```
    print(cities)
    cities = cities + ["Birmingham", "Bradford"]
    print(cities)
locations = ["London", "Leeds", "Glasgow", "Sheffield"]
no_side_effects(locations)
```

```
['London', 'Leeds', 'Glasgow', 'Sheffield']
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham', 'Bradfor
d']
```

```
print(locations)
```

```
['London', 'Leeds', 'Glasgow', 'Sheffield']
```

This changes drastically, if we increment the list by using augmented assignment operator +=. To show this, we change the previous function rename it as "side_effects" in the following example:

```
def side_effects(cities):
    print(cities)
    cities += ["Birmingham", "Bradford"]
    print(cities)

locations = ["London", "Leeds", "Glasgow", "Sheffield"]
side_effects(locations)
```

```
['London', 'Leeds', 'Glasgow', 'Sheffield']
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham', 'Bradfor
d']
```

```
print(locations)
```

```
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham', 'Bradfor
d']
```

We can see that Birmingham and Bradford are included in the global list locations as well, because += acts as an in-place operation.

The user of this function can prevent this side effect by passing a copy to the function. A shallow copy is sufficient, because there are no nested structures in the list. To satisfy our French customers as well, we change the city names in the next example to demonstrate the effect of the slice operator in the function call:

```
def side_effects(cities):
    print(cities)
    cities += ["Paris", "Marseille"]
    print(cities)
```

```
locations = ["Lyon", "Toulouse", "Nice", "Nantes", "Strasbourg"]
side_effects(locations[:])
print(locations)
```

```
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Paris', 'Mar
seille']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
```

```
print(locations)
```

```
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
```

We can see that the global list locations has not been effected by the execution of the function.

COMMAND LINE ARGUMENTS

If you use a command line interface, i.e. a text user interface (TUI) , and not a graphical user interface (GUI), command line arguments are very useful. They are arguments which are added after the function call in the same line.

It's easy to write Python scripts using command line arguments. If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces. Inside the script these arguments are accessible through the list variable sys.argv. The name of the script is included in this list sys.argv[0]. sys.argv[1] contains the first parameter, sys.argv[2] the second and so on. The following script (arguments.py) prints all arguments:

```
  # Module sys has to be imported:
import sys
```

ITERATION OVER ALL ARGUMENTS:

```
for eachArg in sys.argv:
        print(eachArg)
```

Example call to this script:

```
python argumente.py python course for beginners
```

This call creates the following output:

```
argumente.py
python
course
for
```

VARIABLE LENGTH OF PARAMETERS

We will introduce now functions, which can take an arbitrary number of arguments. Those who have some programming background in C or C++ know this from the varargs feature of these languages.

Some definitions, which are not really necessary for the following: A function with an arbitrary number of arguments is usually called a variadic function in computer science. To use another special term: A variadic function is a function of indefinite arity. The arity of a function or an operation is the number of arguments or operands that the function or operation takes. The term was derived from words like "unary", "binary", "ternary", all ending in "ary".

The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

```python
def varpafu(*x):
    print(x)
varpafu()
```

```
()
```

```python
varpafu(34,"Do you like Python?", "Of course")
```

```
(34, 'Do you like Python?', 'Of course')
```

We learn from the previous example that the arguments passed to the function call of varpafu() are collected in a tuple, which can be accessed as a "normal" variable x within the body of the function. If the function is called without any arguments, the value of x is an empty tuple.

Sometimes, it's necessary to use positional parameters followed by an arbitrary number of parameters in a function definition. This is possible, but the positional parameters always have to precede the arbitrary parameters. In the following example, we have a positional parameter "city", - the main location, - which always have to be given, followed by an arbitrary number of other locations:

```python
def locations(city, *other_cities): print(city, other_cities)
locations("Paris")
locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux", "Mar
seille")
```

```
Paris ()
Paris ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille')
```

* IN FUNCTION CALLS

A * can appear in function calls as well, as we have just seen in the previous exercise: The semantics is in this case "inverse" to a star in a function definition. An argument will be unpacked and not packed. In other words, the elements of the list or tuple are singularized:

```
def f(x,y,z):
    print(x,y,z)

p = (47,11,12)
f(*p)
```

```
47 11 12
```

There is hardly any need to mention that this way of calling our function is more comfortable than the following one:

```
f(p[0],p[1],p[2])
```

```
47 11 12
```

Additionally, the previous call (f(p[0],p[1],p[2])) doesn't work in the general case, i.e. lists of unknown lengths. "Unknown" means that the length is only known at runtime and not when we are writing the script.

ARBITRARY KEYWORD PARAMETERS

There is also a mechanism for an arbitrary number of keyword parameters. To do this, we use the double asterisk "**" notation:

```
>>> def f(**args):
    print(args)

f()
```

```
{}
```

```
f(de="German",en="English",fr="French")
```

```
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

DOUBLE ASTERISK IN FUNCTION CALLS

The following example demonstrates the usage of ** in a function call:

```
def f(a,b,x,y):
```

```
    print(a,b,x,y)

d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

and now in combination with *:

```
t = (47,11)
d = {'x':'extract','y':'yes'}
f(*t, **d)
```

```
47 11 extract yes
```

Write a function which calculates the arithmetic mean of a variable number of values.

```
def arithmetic_mean(x, *l):
    """ The function calculates the arithmetic mean of a non-empty
        arbitrary number of numbers """
    sum = x
    for i in l:
        sum += i

    return sum / (1.0 + len(l))
```

You might ask yourself, why we used both a positional parameter "x" and the variable parameter "*l*" *in our function definition. We could have only used* l to contain all our numbers. We wanted to enforce that we always have a non-empty list of numbers. This is necessary to prevent a division by zero error, because the average of an empty list of numbers is not defined.

In the following interactive Python session, we can learn how to use this function. We assume that the function arithmetic_mean is saved in a file called statistics.py.

```
from statistics import arithmetic_mean
arithmetic_mean(4,7,9)
6.666666666666667
arithmetic_mean(4,7,9,45,-3.7,99)
26.71666666666667
```

This works fine, but there is a catch. What if somebody wants to call the function with a list, instead of a variable number of numbers, as we have shown above? We can see in the following that we raise an error, as most hopefully, you might expect:

```
l = [4,7,9,45,-3.7,99]
arithmetic_mean(l)
```

```
---------------------------------------------------------------------
NameError                                   Traceback (most recent call last)
<ipython-input-29-b0693d70ca11> in <module>
      1 l = [4,7,9,45,-3.7,99]
----> 2 arithmetic_mean(l)

NameError: name 'arithmetic_mean' is not defined
```

The rescue is using another asterisk:
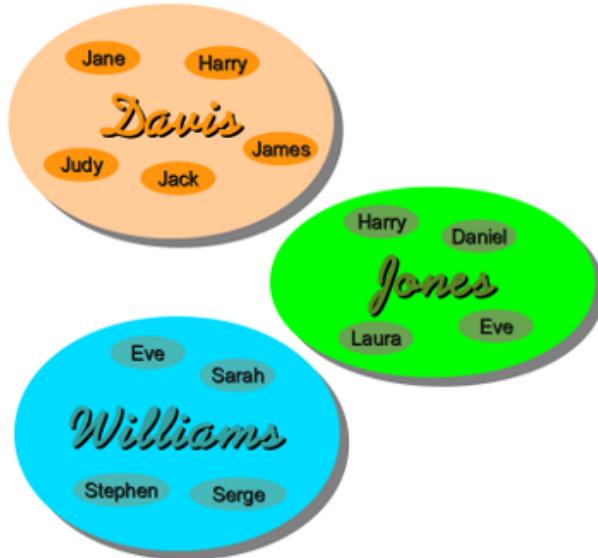
```
arithmetic_mean(*l)
26.71666666666667
```

NAMESPACES AND SCOPES

NAMESPACES

Generally speaking, a namespace (sometimes also called a context) is a naming system for making names unique to avoid ambiguity. Everybody knows a namespacing system from daily life, i.e. the naming of people by the first name and family name (surname). Another example is a network: each network device (workstation, server, printer, ...) needs a unique name and address. Yet a further example is the directory structure of filesystems. The same filename can be used in different directories, the files can be uniquely accessed via the pathnames. Many programming languages use namespaces or contexts for identifiers. An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces (Like the same filename in different directories). Programming languages, which support namespaces, may have different rules that determine to which namespace an identifier belongs to. Namespaces in Python are implemented as Python dictionaries, that is, they are defined by a mapping of names, i.e. the keys of the dictionary, to objects, i.e. the values. The user doesn't have to know this to write a Python program and when using namespaces. Some namespaces in Python:

- *global names* of a module
- *local names* in a function or method invocation
- *built-in names*: this namespace contains built-in fuctions (e.g. abs(), cmp(), ...) and built-in exception names

LIFETIME OF A NAMESPACE

Not every namespace, which may be used in a script or program is accessible (or alive) at any moment during the execution of the script. Namespaces have different lifetimes, because they are often created at different points in time. There is one namespace which is present from beginning to end: The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace of a module is generated when the module is read in. Module namespaces normally last until the script ends, i.e. the interpreter quits. When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

SCOPES

A scope refers to a region of a program where a namespace can be directly accessed, i.e. without using a

namespace prefix. In other words, the scope of a name is the area of a program where this name can be unambiguously used, for example, inside of a function. A name's namespace is identical to it's scope. Scopes are defined statically, but they are used dynamically. During program execution there are the following nested scopes available:

- the innermost scope is searched first and it contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope
- the next-to-last scope contains the current module's global names
- the outermost scope, which is searched last, is the namespace containing the built-in names

GLOBAL, LOCAL AND NONLOCAL VARIABLES

The way Python uses global and local variables is maverick. While in many or most other programming languages variables are treated as global if not declared otherwise, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out. Like in many other program structures, Python also imposes good programming habit by design.

So when you define variables inside a function definition, they are local to this function by default. That is, anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. In other words, the function body is the scope of such a variable, i.e. the enclosing context where this name is associated with its values.

All variables have the scope of the block, where they are declared and defined. They can only be used after the point of their declaration.

Just to make things clear: Variables don't have to be and can't be declared in the way they are declared in programming languages like Java or C. Variables in Python are implicitly declared by defining them, i.e. the first time you assign a value to a variable, this variable is declared and has automatically the data type of the object which has to be assigned to it. If you have problems understanding this, please consult our chapter about data types and variables, see links on the left side.

GLOBAL AND LOCAL VARIABLES IN FUNCTIONS

In the following example, we want to demonstrate, how global values can be used inside the body of a function:

```python
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

```
I love Paris in the summer!
```

The variable s is defined as the string "I love Paris in the summer!", before calling the function f(). The body of f() consists solely of the "print(s)" statement. As there is no local variable s, i.e. no assignment to s, the value from the global variable s will be used. So the output will be the string "I love Paris in the summer!". The question is, what will happen, if we change the value of s inside of the function f()? Will it affect the global variable as well? We test this in the following piece of code:

```python
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
print(s)
```

```
I love London!
I love Paris!
```

What if we combine the first example with the second one, i.e. we first access s with a print() function, hoping to get the global value, and then assigning a new value to it? Assigning a value to it, means - as we have previously stated - creating a local variable s. So, we would have s both as a global and a local variable in the same scope, i.e. the body of the function. Python fortunately doesn't allow this ambiguity. So, it will raise an error, as we can see in the following example:

```python
def f():
    print(s)
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-3-d7a23bc83c27> in <module>
      5
      6 s = "I love Paris!"
----> 7 f()

<ipython-input-3-d7a23bc83c27> in f()
      1 def f():
----> 2     print(s)
      3     s = "I love London!"
      4     print(s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

A variable can't be both local and global inside a function. So Python decides that we want a local variable due to the assignment to s inside f(), so the first print statement before the definition of s throws the error message above. Any variable which is changed or created inside a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global", as can be seen in the following example:

```python
def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)


s = "I am looking for a course in Paris!"
f()
print(s)
```

```
I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

Local variables of functions can't be accessed from outside, when the function call has finished. Here is the continuation of the previous example:

```python
def f():
    s = "I am globally not known"
    print(s)

f()
```

```
print(s)
```

```
I am globally not known
Only in spring, but London is great as well!
```

The following example shows a wild combination of local and global variables and function parameters:

```python
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)
```

```
42 17 4 17
42 15 3 4
```

GLOBAL VARIABLES IN NESTED FUNCTIONS

We will examine now what will happen, if we use the global keyword inside nested functions. The following example shows a situation where a variable 'city' is used in various scopes:

```python
def f():
    city = "Hamburg"
    def g():
        global city
        city = "Geneva"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

f()
print("Value of city in main: " + city)
```

```
Before calling g: Hamburg
Calling g now:
After calling g: Hamburg
Value of city in main: Geneva
```

We can see that the global statement inside the nested function g does not affect the variable 'city' of the function f, i.e. it keeps its value 'Hamburg'. We can also deduce from this example that after calling f() a variable 'city' exists in the module namespace and has the value 'Geneva'. This means that the global keyword in nested functions does not affect the namespace of their enclosing namespace! This is consistent to what we have found out in the previous subchapter: A variable defined inside of a function is local unless it is explicitly marked as global. In other words, we can refer to a variable name in any enclosing scope, but we can only rebind variable names in the local scope by assigning to it or in the module-global scope by using a global declaration. We need a way to access variables of other scopes as well. The way to do this are nonlocal definitions, which we will explain in the next chapter.

NONLOCAL VARIABLES

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. One difference to global variables lies in the fact that it is not possible to change variables from the module scope, i.e. variables which are not defined inside of a function, by using the nonlocal statement. We show this in the two following examples:

```python
def f():
    global city
    print(city)


city = "Frankfurt"
f()
```

```
Frankfurt
```

This program is correct and returns 'Frankfurt' as the output. We will change "global" to "nonlocal" in the following program:

```python
def f():
    nonlocal city
    print(city)


city = "Frankfurt"
f()
```

```
  File "<ipython-input-9-97bb311dfb80>", line 2
    nonlocal city
    ^
SyntaxError: no binding for nonlocal 'city' found
```

This shows that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

```python
def f():
    city = "Munich"
    def g():
        nonlocal city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
Before calling g: Munich
Calling g now:
After calling g: Zurich
'city' in main: Stuttgart
```

In the previous example the variable 'city' was defined prior to the call of g. We get an error if it isn't defined:

```python
def f():
    #city = "Munich"
    def g():
        nonlocal city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
  File "<ipython-input-11-5417be93b6a6>", line 4
    nonlocal city
    ^
SyntaxError: no binding for nonlocal 'city' found
```

The program works fine - with or without the line 'city = "Munich"' inside of f - , if we change "nonlocal" to "global":

```python
def f():
```

```python
    #city = "Munich"
    def g():
        global city
        city = "Zurich"
    print("Before calling g: " + city)
    print("Calling g now:")
    g()
    print("After calling g: " + city)

city = "Stuttgart"
f()
print("'city' in main: " + city)
```

```
Before calling g: Stuttgart
Calling g now:
After calling g: Zurich
'city' in main: Zurich
```

Yet there is a huge difference: The value of the global x is changed now!

DECORATORS
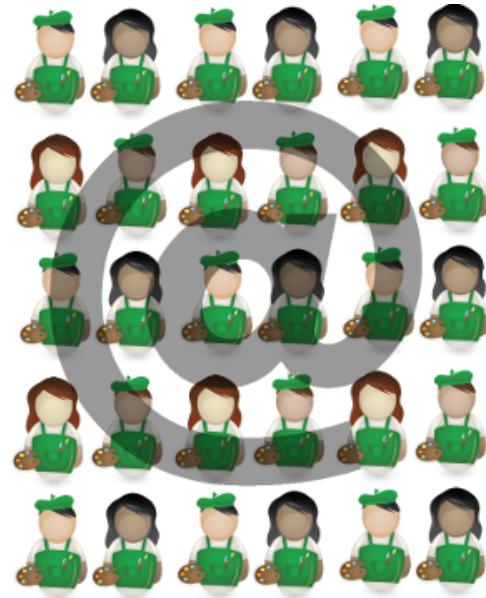
INTRODUCTION

Decorators belong most probably to the most beautiful and most powerful design possibilities in Python, but at the same time the concept is considered by many as complicated to get into. To be precise, the usage of decorators is very easy, but writing decorators can be complicated, especially if you are not experienced with decorators and some functional programming concepts.

Even though it is the same underlying concept, we have two different kinds of decorators in Python:

- Function decorators
- Class decorators

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

You may also consult our chapter on memoization with decorators.

If you like the image on the right side of this page and if you are also interested in image processing with Python, Numpy, Scipy and Matplotlib, you will definitely like our chapter on Image Processing Techniques, it explains the whole process of the making-of of our decorator and at sign picture!

FIRST STEPS TO DECORATORS

We know from our various Python training classes that there are some points in the definitions of decorators, where many beginners get stuck.

Therefore, we will introduce decorators by repeating some important aspects of functions. First you have to know or remember that function names are references to functions and that we can assign multiple names to the same function:

```python
def succ(x):
    return x + 1
successor = succ
successor(10)
```

Output: 11

```python
succ(10)
```

This means that we have two names, i.e. "succ" and "successor" for the same function. The next important fact is that we can delete either "succ" or "successor" without deleting the function itself.

```python
del succ
successor(10)
```

The concept of having or defining functions inside of a function is completely new to C or C++ programmers:

```python
def f():

    def g():
        print("Hi, it's me 'g'")
        print("Thanks for calling me")

    print("This is the function 'f'")
    print("I am calling 'g' now:")
    g()


f()
```

```
This is the function 'f'
I am calling 'g' now:
Hi, it's me 'g'
Thanks for calling me
```

Another example using "proper" return statements in the functions:

```python
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

print(temperature(20))
```

```
It's 68.0 degrees!
```

The following example is about the factorial function, which we previously defined as follows:

```python
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

What happens if someone passes a negative value or a float number to this function? It will never end. You might get the idea to check that as follows:

```python
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    if type(n) == int and n >=0:
        if n == 0:
            return 1
        else:
            return n * factorial(n-1)

    else:
        raise TypeError("n has to be a positive integer or zero")
```

If you call this function with `4 '' for example, i.e.` factorial (4) ", the first thing that is checked is whether it is my positive integer. In principle, this makes sense. The "problem" now appears in the recursion step. Now `factorial (3) '' is called. This call and all others also check whether it is a positive whole number. But this is unnecessary: If you subtract the value` 1 " from a positive whole number, you get a positive whole number or `` 0 " again. So both well-defined argument values for our function.

With a nested function (local function) one can solve this problem elegantly:

```python
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    def inner_factorial(n):
        if n == 0:
            return 1
        else:
            return n * inner_factorial(n-1)
    if type(n) == int and n >=0:
        return inner_factorial(n)
```

```
    else:
        raise TypeError("n should be a positve int or 0")
```

In [ ]:

If you solely look at the previous examples, this doesn't seem to be very useful. It gets useful in combination with two further powerful possibilities of Python functions. Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

```python
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

```
Hi, it's me 'f'
I will call 'func' now
Hi, it's me 'g'
Thanks for calling me
```

You may not be satisfied with the output. 'f' should write that it calls 'g' and not 'func'. Of course, we need to know what the 'real' name of func is. For this purpose, we can use the attribute __name__, as it contains this name:

```python
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)


f(g)
```

```
Hi, it's me 'f'
I will call 'func' now
Hi, it's me 'g'
Thanks for calling me
func's real name is g
```

The output explains what's going on once more. Another example:

```python
import math

def foo(func):
    print("The function " + func.__name__ + " was passed to foo")
    res = 0
    for x in [1, 2, 2.5]:
        res += func(x)
    return res

print(foo(math.sin))
print(foo(math.cos))
```

```
The function sin was passed to foo
2.3492405557375347
The function cos was passed to foo
-0.6769881462259364
```

The output of a function is also a reference to an object. Therefore functions can return references to function objects.

```python
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

```
5
7
```

The previous example looks very artificial and absolutely useless. We will present now another language oriented example, which shows a more practical touch. Okay, still not a function which is useful the way it is. We write a function with the nearly self-explanatory name `greeting_func_gen`. So this function returns (or generates) functions which can be used to create people in different languages, i.e. German, French, Italian, Turkish, and Greek:

```python
def greeting_func_gen(lang):
    def customized_greeting(name):
        if lang == "de":    # German
            phrase = "Guten Morgen "
        elif lang == "fr": # French
            phrase = "Bonjour "
        elif lang == "it": # Italian
            phrase = "Buongiorno "
        elif lang == "tr": # Turkish
            phrase = "Günaydın "
        elif lang == "gr": # Greek
            phrase = "Καλημερα "
        else:
            phrase = "Hi "
        return phrase + name + "!"
    return customized_greeting


say_hi = greeting_func_gen("tr")
print(say_hi("Gülay"))      # this Turkish name means "rose moon" b
y the way
```

```
Günaydın Gülay!
```

It is getting more useful and at the same time more mathematically oriented in the following example. We will implement a polynomial "factory" function now. We will start with writing a version which can create polynomials of degree 2.

$$p(x) = ax^2 + bx + c$$

The Python implementation as a polynomial factory function can be written like this:

```python
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x**2 + b * x + c
    return polynomial

p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)
```

```
for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

```
-2 1 -7
-1 -2 -2
0 -1 1
1 4 2
```

We can generalize our factory function so that it can work for polynomials of arbitrary degree:

$$\sum_{k=0}^{n} a_k \cdot x^k = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

```python
def polynomial_creator(*coefficients):
    """ coefficients are in the form a_n, ... a_1, a_0
    """
    def polynomial(x):
        res = 0
        for index, coeff in enumerate(coefficients[::-1]):
            res += coeff * x** index
        return res
    return polynomial

p1 = polynomial_creator(4)
p2 = polynomial_creator(2, 4)
p3 = polynomial_creator(1, 8, -1, 3, 2)
p4  = polynomial_creator(-1, 2, 1)


for x in range(-2, 2, 1):
    print(x, p1(x), p2(x), p3(x), p4(x))
```

```
-2 4 0 -56 -7
-1 4 2 -9 -2
0 4 4 2 1
1 4 6 13 2
```

The function p3 implements, for example, the following polynomial:

$$p_3(x) = x^4 + 8 \cdot x^3 - x^2 + 3 \cdot x + 2$$

The polynomial function inside of our decorator polynomial_creator can be implemented more efficiently. We

can factorize it in a way so that it doesn't need any exponentiation.

Factorized version of a general polynomial without exponentiation:

$$res = (\dots (a_n \cdot x + a_{n-1}) \cdot x + \dots + a_1) \cdot x + a_0$$

Implementation of our polynomial creator decorator avoiding exponentiation:

```python
def polynomial_creator(*coeffs):
    """ coefficients are in the form a_n, a_n_1, ... a_1, a_0
    """
    def polynomial(x):
        res = coeffs[0]
        for i in range(1, len(coeffs)):
            res = res * x + coeffs[i]
        return res

    return polynomial

p1 = polynomial_creator(4)
p2 = polynomial_creator(2, 4)
p3 = polynomial_creator(1, 8, -1, 3, 2)
p4 = polynomial_creator(-1, 2, 1)


for x in range(-2, 2, 1):
    print(x, p1(x), p2(x), p3(x), p4(x))
```

```
-2 4 0 -56 -7
-1 4 2 -9 -2
0 4 4 2 1
1 4 6 13 2
```

If you want to learn more about polynomials and how to create a polynomial class, you can continue with our chapter on Polynomials.

A SIMPLE DECORATOR

Now we have everything ready to define our first simple decorator:

```python
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
```

```python
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

```
We call foo before decoration:
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 42
After calling foo
```

If you look at the output of the previous program, you can see what's going on. After the decoration "foo = our_decorator(foo)", foo is a reference to the 'function_wrapper'. 'foo' will be called inside of 'function_wrapper', but before and after the call some additional code will be executed, i.e. in our case two print functions.

THE USUAL SYNTAX FOR DECORATORS IN PYTHON

The decoration in Python is usually not performed in the way we did it in our previous example, even though the notation `foo = our_decorator(foo)` is catchy and easy to grasp. This is the reason, why we used it! You can also see a design problem in our previous approach. "foo" existed in the same program in two versions, before decoration and after decoration.

We will do a proper decoration now. The decoration occurrs in the line before the function header. The "@" is followed by the decorator function name.

We will rewrite now our initial example. Instead of writing the statement

```
    foo = our_decorator(foo)
```

we can write

```
    @our_decorator
```

But this line has to be directly positioned in front of the decorated function. The complete example looks like

this now:

```python
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

```
Before calling foo
Hi, foo has been called with Hi
After calling foo
```

We can decorate every other function which takes one parameter with our decorator 'our_decorator'. We demonstrate this in the following. We have slightly changed our function wrapper, so that we can see the result of the function calls:

```python
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def succ(n):
    return n + 1

succ(10)
```

```
Before calling succ
11
After calling succ
```

It is also possible to decorate third party functions, e.g. functions we import from a module. We can't use the Python syntax with the "at" sign in this case:

```python
from math import sin, cos
```

```python
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

```
Before calling sin
9.265358966049026e-05
After calling sin
Before calling cos
-0.9999999957076562
After calling cos
```

All in all, we can say that a decorator in Python is a callable Python object that is used to modify a function, method or class definition. The original object, the one which is going to be modified, is passed to a decorator as an argument. The decorator returns a modified object, e.g. a modified function, which is bound to the name used in the definition.

The previous function_wrapper works only for functions with exactly one parameter. We provide a generalized version of the function_wrapper, which accepts functions with arbitrary parameters in the following example:

```python
from random import random, randint, choice

def our_decorator(func):
    def function_wrapper(*args, **kwargs):
        print("Before calling " + func.__name__)
        res = func(*args, **kwargs)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

random = our_decorator(random)
randint = our_decorator(randint)
choice = our_decorator(choice)
```

```
random()
randint(3, 8)
choice([4, 5, 6])
```

```
Before calling random
0.3206237466802222
After calling random
Before calling randint
6
After calling randint
Before calling choice
5
After calling choice
```

USE CASES FOR DECORATORS

CHECKING ARGUMENTS WITH A DECORATOR

In our chapter about recursive functions we introduced the factorial function. We wanted to keep the function as simple as possible and we didn't want to obscure the underlying idea, so we didn't incorporate any argument checks. So, if somebody called our function with a negative argument or with a float argument, our function would get into an endless loop.

The following program uses a decorator function to ensure that the argument passed to the function factorial is a positive integer:

```python
def argument_test_natural_number(f):
    def helper(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
        print(i, factorial(i))

print(factorial(-1))
```

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
```

```
------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-18-25b69b5bbe5f> in <module>
     17             print(i, factorial(i))
     18
---> 19 print(factorial(-1))

<ipython-input-18-25b69b5bbe5f> in helper(x)
      4                 return f(x)
      5             else:
----> 6                 raise Exception("Argument is not an integer")
      7         return helper
      8

Exception: Argument is not an integer
```

The following example uses a decorator to count the number of times a function has been called. To be precise, we can use this decorator solely for functions with exactly one parameter:

```python
def call_counter(func):
    def helper(x):
        helper.calls += 1
        return func(x)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

print(succ.calls)
for i in range(10):
```

```
    succ(i)

print(succ.calls)
```

```
0
10
```

We pointed out that we can use our previous decorator only for functions, which take exactly one parameter. We will use the *args and **kwargs notation to write decorators which can cope with functions with an arbitrary number of positional and keyword parameters.

```python
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

@call_counter
def mul1(x, y=1):
    return x*y + 1

print(succ.calls)
for i in range(10):
    succ(i)
mul1(3, 4)
mul1(4)
mul1(y=3, x=2)

print(succ.calls)
print(mul1.calls)
```

```
0
10
3
```

DECORATORS WITH PARAMETERS

We define two decorators in the following code:

```
def evening_greeting(func):
    def function_wrapper(x):
        print("Good evening, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper

def morning_greeting(func):
    def function_wrapper(x):
        print("Good morning, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper

@evening_greeting
def foo(x):
    print(42)

foo("Hi")
```

```
Good evening, foo returns:
42
```

These two decorators are nearly the same, except for the greeting. We want to add a parameter to the decorator to be capable of customizing the greeting, when we do the decoration. We have to wrap another function around our previous decorator function to accomplish this. We can now easily say "Good Morning" in Greek:

```
def greeting(expr):
    def greeting_decorator(func):
        def function_wrapper(x):
            print(expr + ", " + func.__name__ + " returns:")
            func(x)
        return function_wrapper
    return greeting_decorator

@greeting("καλημερα")
def foo(x):
    print(42)

foo("Hi")
```

```
καλημερα, foo returns:
42
```

If we don't want or cannot use the "at" decorator syntax, we can do it with function calls:

```python
def greeting(expr):
    def greeting_decorator(func):
        def function_wrapper(x):
            print(expr + ", " + func.__name__ + " returns:")
            return func(x)
        return function_wrapper
    return greeting_decorator


def foo(x):
    print(42)

greeting2 = greeting("καλημερα")
foo = greeting2(foo)
foo("Hi")
```

```
καλημερα, foo returns:
42
```

Of course, we don't need the additional definition of "greeting2". We can directly apply the result of the call "greeting("καλημερα")" on "foo":

```
foo = greeting("καλημερα")(foo)
```

USING WRAPS FROM FUNCTOOLS

The way we have defined decorators so far hasn't taken into account that the attributes

- __name__ (name of the function),
- __doc__ (the docstring) and
- __module__ (The module in which the function is defined)

of the original functions will be lost after the decoration.

The following decorator will be saved in a file greeting_decorator.py:

```python
def greeting(func):
def function_wrapper(x):
    """ function_wrapper of greeting """
    print("Hi, " + func.__name__ + " returns:")
    return func(x)
return function_wrapper
```

We call it in the following program:

```python
from greeting_decorator import greeting

@greeting
def f(x):
    """ just some silly function """
    return x + 4

f(10)
print("function name: " + f.__name__)
print("docstring: " + f.__doc__)
print("module name: " + f.__module__)
```

```
Hi, f returns:
function name: function_wrapper
docstring:  function_wrapper of greeting
module name: greeting_decorator
```

We get the "unwanted" results above.

We can save the original attributes of the function f, if we assign them inside of the decorator. We change our previous decorator accordingly and save it as greeting_decorator_manually.py:

```python
def greeting(func):
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
    function_wrapper.__name__ = func.__name__
    function_wrapper.__doc__ = func.__doc__
    function_wrapper.__module__ = func.__module__
    return function_wrapper
```

In our main program, all we have to do is change the import statement.

```python
from greeting_decorator_manually import greeting
```

Fortunately, we don't have to add all this code to our decorators to have these results. We can import the decorator "wraps" from functools instead and decorate our function in the decorator with it:

```python
from functools import wraps
```

```
def greeting(func):
    @wraps(func)
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper
```

CLASSES INSTEAD OF FUNCTIONS

THE CALL METHOD

So far we used functions as decorators. Before we can define a decorator as a class, we have to introduce the `__call__` method of classes. We mentioned already that a decorator is simply a callable object that takes a function as an input parameter. A function is a callable object, but lots of Python programmers don't know that there are other callable objects. A callable object is an object which can be used and behaves like a function but might not be a function. It is possible to define classes in a way that the instances will be callable objects. The `__call__` method is called, if the instance is called "like a function", i.e. using brackets.

```
class A:
    def __init__(self):
        print("An instance of A was initialized")

    def __call__(self, *args, **kwargs):
        print("Arguments are:", args, kwargs)


x = A()
print("now calling the instance:")
x(3, 4, x=11, y=10)
print("Let's call it again:")
x(3, 4, x=11, y=10)
```

```
An instance of A was initialized
now calling the instance:
Arguments are: (3, 4) {'x': 11, 'y': 10}
Let's call it again:
Arguments are: (3, 4) {'x': 11, 'y': 10}
```

We can write a class for the fibonacci function by using the `__call__` method:

```
class Fibonacci:

    def __init__(self):
```

```
        self.cache = {}

    def __call__(self, n):
        if n not in self.cache:
            if n == 0:
                self.cache[0] = 0
            elif n == 1:
                self.cache[1] = 1
            else:
                self.cache[n] = self.__call__(n-1) + self.__cal
l__(n-2)
        return self.cache[n]

fib = Fibonacci()

for i in range(15):
    print(fib(i), end=", ")
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
```

You can find further information on the `__call__` method in the chapter Magic Functions of our tutorial.

We will rewrite the following decorator as a class:

```
def decorator1(f):
    def helper():
        print("Decorating", f.__name__)
        f()
    return helper

@decorator1
def foo():
    print("inside foo()")

foo()
```

```
Decorating foo
inside foo()
```

The following decorator implemented as a class does the same "job":

```
class decorator2:
```

```python
    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```
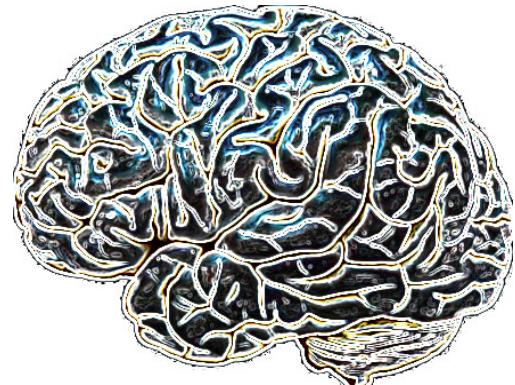
```
Decorating foo
inside foo()
```

Both versions return the same output.

MEMOIZATION WITH DECORATORS

DEFINITION OF MEMOIZATION

The term "memoization" was introduced by Donald Michie in the year 1968. It's based on the Latin word memorandum, meaning "to be remembered". It's not a misspelling of the word memorization, though in a way it has something in common. Memoisation is a technique used in computing to speed up programs. This is accomplished by memorizing the calculation results of processed input such as the results of function calls. If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided. In many cases a simple array is used for storing the results, but lots of other structures can be used as well, such as associative arrays, called hashes in Perl or dictionaries in Python.

Memoization can be explicitly programmed by the programmer, but some programming languages like Python provide mechanisms to automatically memoize functions.

MEMOIZATION WITH FUNCTION DECORATORS

You may consult our chapter on decorators as well. Especially, if you may have problems in understanding our reasoning.

In our previous chapter about recursive functions, we worked out an iterative and a recursive version to calculate the Fibonacci numbers. We have shown that a direct implementation of the mathematical definition into a recursive function like the following has an exponential runtime behaviour:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

We also presented a way to improve the runtime behaviour of the recursive version by adding a dictionary to memorize previously calculated values of the function. This is an example of explicitly using the technique of memoization, but we didn't call it like this. The disadvantage of this method is that the clarity and the beauty of the original recursive implementation is lost.

The "problem" is that we changed the code of the recursive fib function. The following code doesn't change our fib function, so that its clarity and legibility isn't touched. To this purpose, we define and use a function which we call memoize. memoize() takes a function as an argument. The function memoize uses a dictionary "memo" to store the function results. Though the variable "memo" as well as the function "f" are local to memoize, they are captured by a closure through the helper function which is returned as a reference by memoize(). So, the call memoize(fib) returns a reference to the helper() which is doing what fib() would do on its own plus a wrapper which saves the calculated results. For an integer 'n' fib(n) will only be called, if n is not in the memo dictionary. If it is in it, we can output memo[n] as the result of fib(n).

```python
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper


def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib = memoize(fib)

print(fib(40))
```

```
102334155
```

Let's look at the line in our code where we call memoize with fib as the argument:

```
fib = memoize(fib)
```

Doing this, we turn memoize into a decorator. One says that the fib function is decorated by the memoize() function. We will illustrate with the following diagrams how the decoration is accomplished. The first diagram illustrates the state before the decoration, i.e. before we call fib = memoize(fib). We can see the function names referencing their bodies:

After having executed fib = memoize(fib) fib points to the body of the helper function, which had been returned by memoize. We can also perceive that the code of the original fib function can only be reached via the "f" function of the helper function from now on. There is no other way anymore to call the original fib directly, i.e. there is no other reference to it. The decorated Fibonacci function is called in the return statement return fib(n-1) + fib(n-2), this means the code of the helper function which had been returned by memoize:

```
memoize

memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper
```

Another point in the context of decorators deserves special attention: We don't usually write a decorator for just one use case or function. We rather use it multiple times for different functions. So we could imagine having further functions func1, func2, func3 and so on, which consume also a lot of time. Therefore, it makes sense to decorate each one with our decorator function "memoize":

```
fib

if n == 0:
    return 0
elif n == 1:
    return 1
else:
    return fib(n-1) + fib(n-2)
```

```
fib   = memoize(fib)
func1 = memoize(func1)
func2 = memoize(func2)
func3 = memoize(func3)
# and so on
```

```
memoize

memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper
```

Executing:
fib = memoize(fib)
helper is returned

We haven't used the Pythonic way of writing a decorator. Instead of writing the statement
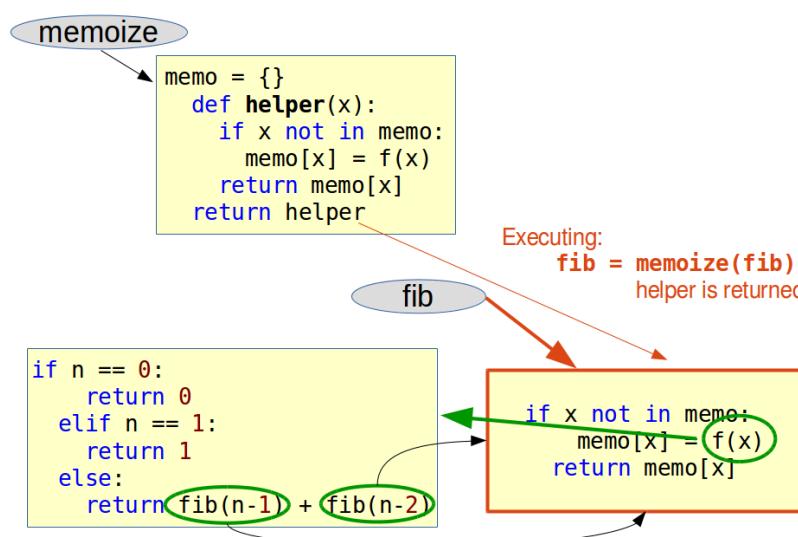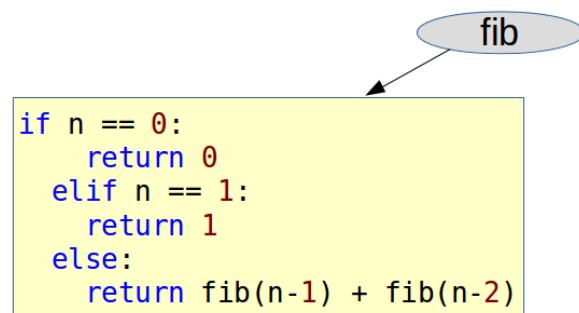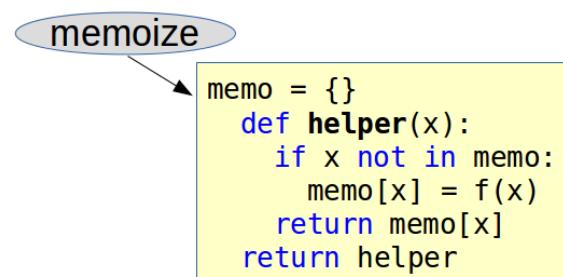
```
fib   = memoize(fib)
```

we should have "decorated" our fib function with:

```
fib

if n == 0:
    return 0
elif n == 1:
    return 1
else:
    return fib(n-1) + fib(n-2)
```

```
if x not in memo:
    memo[x] = f(x)
return memo[x]
```

```
@memoize
```

But this line has to be directly in front of the decorated function, in our example fib(). The complete example in a Pythonic way looks like this now:

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
```

```python
    return helper

@memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(40))
```

```
102334155
```

USING A CALLABLE CLASS FOR MEMOIZATION

This subchapter can be skipped without problems by those who don't know about object orientation so far.

We can encapsulate the caching of the results in a class as well, as you can see in the following example:

```python
class Memoize:

    def __init__(self, fn):
        self.fn = fn
        self.memo = {}

    def __call__(self, *args):
        if args not in self.memo:
            self.memo[args] = self.fn(*args)
        return self.memo[args]

@Memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
print(fib(40))
```

```
102334155
```

As we are using a dictionary, we can't use mutable arguments, i.e. the arguments have to be immutable.

EXERCISE

- Our exercise is an old riddle, going back to 1612. The French
  Jesuit Claude-Gaspar Bachet phrased it. We have to weigh
  quantities (e.g. sugar or flour) from 1 to 40 pounds. What is
  the least number of weights that can be used on a balance
  scale to way any of these quantities?

The first idea might be to use weights of 1, 2, 4, 8, 16 and 32 pounds. This
is a minimal number, if we restrict ourself to put weights on one side and
the stuff, e.g. the sugar, on the other side. But it is possible to put weights
on both pans of the scale. Now, we need only four weights, i.e. 1, 3, 9, 27

Write a Python function weigh(), which calculates the weights needed and their distribution on the pans to
weigh any amount from 1 to 40.

SOLUTION

We need the function linear_combination() from our chapter [Linear Combinations](#).

```python
def factors_set():
        for i in [-1, 0, 1]:
            for j in [-1,0,1]:
                for k in [-1,0,1]:
                    for l in [-1,0,1]:
                        yield (i, j, k, l)

def memoize(f):
    results = {}
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper

@memoize
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27      """
    weighs = (1,3,9,27)

    for factors in factors_set():
        sum = 0
        for i in range(len(factors)):
```

```
            sum += factors[i] * weighs[i]
        if sum == n:
            return factors
```

With this, it is easy to write our function weigh().

```
def weigh(pounds):
        weights = (1, 3, 9, 27)
        scalars = linear_combination(pounds)
        left = ""
        right = ""
        for i in range(len(scalars)):
            if scalars[i] == -1:
                left += str(weights[i]) + " "
            elif scalars[i] == 1:
                right += str(weights[i]) + " "
        return (left,right)

for i in [2, 3, 4, 7, 8, 9, 20, 40]:
        pans = weigh(i)
        print("Left  pan: " + str(i) + " plus " + pans[0])
        print("Right pan: " + pans[1] + "\n")
```

```
Left  pan: 2 plus 1
Right pan: 3

Left  pan: 3 plus
Right pan: 3

Left  pan: 4 plus
Right pan: 1 3

Left  pan: 7 plus 3
Right pan: 1 9

Left  pan: 8 plus 1
Right pan: 9

Left  pan: 9 plus
Right pan: 9

Left  pan: 20 plus 1 9
Right pan: 3 27

Left  pan: 40 plus
Right pan: 1 3 9 27
```

FILE MANAGEMENT

FILES IN GENERAL

It's hard to find anyone in the 21st century, who doesn't know what a file is. When we say file, we mean of course, a file on a computer. There may be people who don't know anymore the "container", like a cabinet or a folder, for keeping papers archived in a convenient order. A file on a computer is the modern counterpart of this. It is a collection of information, which can be accessed and used by a computer program. Usually, a file resides on a durable storage. Durable means that the data is persistent, i.e. it can be used by other programs after the program which has created or manipulated it, has terminated.

The term file management in the context of computers refers to the manipulation of data in a file or files and documents on a computer. Though everybody has an understanding of the term file, we present a formal definition anyway:

A file or a computer file is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

The term "file" - as we have described it in the previous paragraph - appeared in the history of computers very early. Usage can be tracked down to the year 1952, when punch cards where used.

A programming language without the capability to store and retrieve previously stored information would be hardly useful.

The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.

READING AND WRITING FILES IN PYTHON

The syntax for reading and writing files in Python is similar to programming languages like C, C++, Java, Perl, and others but a lot easier to handle.

We will start with writing a file. We have a string which contains part of the definition of a general file from Wikipedia:

```
definition = """
A computer file is a computer resource for recording data discrete
ly in a
computer storage device. Just as words can be written
to paper, so can information be written to a computer
```

```
file. Files can be edited and transferred through the
internet on that particular computer system."""
```

We will write this into a file with the name `file_definition.txt` :

```
open("file_definition.txt", "w").write(definition)
```
Output: 283

If you check in your file browser, you will see a file with this name. The file will look like this:
file_definition.txt

We successfully created and have written to a text file. Now, we want to see how to read this file from Python. We can read the whole text file into one string, as you can see in the following code:

```
text = open("file_definition.txt").read()
```

If you call `print(text)` , you will see the text from above again.

Reading in a text file in one string object is okay, as long as the file is not too large. If a file is large, wwe can read in the file line by line. We demonstrate how this can be achieved in the following example with a small file:

```python
with open("ad_lesbiam.txt", "r") as fh:
    for line in fh:
        print(line.strip())
```

```
V. ad Lesbiam

VIVAMUS mea Lesbia, atque amemus,
rumoresque senum severiorum
omnes unius aestimemus assis!
soles occidere et redire possunt:
nobis cum semel occidit breuis lux,
nox est perpetua una dormienda.
da mi basia mille, deinde centum,
dein mille altera, dein secunda centum,
deinde usque altera mille, deinde centum.
dein, cum milia multa fecerimus,
conturbabimus illa, ne sciamus,
aut ne quis malus inuidere possit,
cum tantum sciat esse basiorum.
(GAIUS VALERIUS CATULLUS)
```

Some people don't use the with statement to read or write files. This is not a good idea. The code above without with looks like this:

```
fh = open("ad_lesbiam.txt")
for line in fh:
        print(line.strip())
fh.close()
```

A striking difference between both implementation consists in the usage of `close` . If we use `with` , we do not have to explicitly close the file. The file will be closed automatically, when the with blocks ends. Without `with` , we have to explicitly close the file, like in our second example with `fh.close()` . There is a more important difference between them: If an exception occurs inside of the `with` block, the file will be closed. If an exception occurs in the variant without `with` before the `close` , the file will not be closed. This means, you should alwawys use the `with` statement.

We saw already how to write into a file with "write". The following code is an example, in which we show how to read in from one file line by line, change the lines and write the changed content into another file. The file can be downloaded: :

```python
with open("pythonista_and_python.txt") as infile:
    with open("python_newbie_and_the_guru.txt", "w") as outfile:
        for line in infile:
            line = line.replace("Pythonista", "Python newbie")
            line = line.replace("Python snake", "Python guru")
            print(line.rstrip())
            # write the line into the file:
            outfile.write(line)
```

```
A blue Python newbie, green behind the ears, went to Pythonia.
She wanted to visit the famous wise green Python guru.
She wanted to ask her about the white way to avoid the black.
The bright path to program in a yellow, green, or blue style.
The green Python turned red, when she addressed her.
The Python newbie turned yellow in turn.
After a long but not endless loop  the wise Python uttered:
"The rainbow!"
```

As we have already mentioned: If a file is not to large and if we have to do replacements like we did in the previous example, we wouldn't read in and write out the file line by line. It is much better to use the `read` method, which returns a string containing the complete content of the file, including the carriage returns and line feeds. We can apply the changes to this string and save it into the new file. Working like this, there is no need for a `with` construct, because there will be no reference to the file, i.e. it will be immediately deleted afeter reading and writing:

```python
txt = open("pythonista_and_python.txt").read()
```

```
txt = txt.replace("Pythonista", "Python newbie")
txt = txt.replace("Python snake", "Python guru")
open("python_newbie_and_the_guru.txt", "w").write(txt)
;
```
Output: ''

RESETTING THE FILES CURRENT POSITION

It's possible to set - or reset - a file's position to a certain position, also called the offset. To do this, we use the method `seek`. The parameter of seek determines the offset which we want to set the current position to. To work with `seek`, we will often need the method `tell` which "tells" us the current position. When we have just opened a file, it will be zero. Before we demonstrate the way of working of both `seek` and `tell`, we create a simple file on which we will perform our commands:

```
open("small_text.txt", "w").write("brown is her favorite colour")
;
```
Output: ''

The method `tell` returns the current stream position, i.e. the position where we will continue, when we use a "read", "readline" or so on:

```
fh = open("small_text.txt")
fh.tell()
```
Output: 0

Zero tells us that we are positioned at the first character of the file.

We will read now the next five characters of the file:

```
fh.read(5)
```
Output: 'brown'

Using `tell` again, shows that we are located at position 5:

```
fh.tell()
```
Output: 5

Using `read` without parameters will read the remainder of the file starting from this position:

```
fh.read()
```

Output: `' is her favorite colour'`

Using `tell` again, tells us about the position after the last character of the file. This number corresponds to the number of characters of the file!

```
fh.tell()
```

Output: `28`

With `seek` we can move the position to an arbitrary place in the file. The method `seek` takes two parameters:

```
fh.seek(offset, startpoint_for_offset)
```

where fh is the file pointer, we are working with. The parameter `offset` specifies how many positions the pointer will be moved. The question is from which position should the pointer be moved. This position is specified by the second parameter `startpoint_for_offset`. It can have the follwoing values:

```
0: reference point is the beginning of the file
1: reference point is the current file position
2: reference point is the end of the file
```

if the `startpoint_for_offset` parameter is not given, it defaults to 0.

WARNING: The values 1 and 2 for the second parameter work only, if the file has been opened for binary reading. We will cover this later!

The following examples, use the default behaviour:

```
fh.seek(13)
print(fh.tell())    # just to show you, what seek did!
fh.read()           # reading the remainder of the file

13
```

Output: `'favorite colour'`

It is also possible to move the position relative to the current position. If we want to move `k` characters to the right, we can just set the argument of `seek` to `fh.tell() + k`

```
k = 6
fh.seek(5)      # setting the position to 5
fh.seek(fh.tell() + k)    #  moving k positions to the right
```

```python
print("We are now at position: ", fh.tell())
```

```
We are now at position:  11
```

`seek` doesn't like negative arguments for the position. On the other hand it doesn't matter, if the value for the position is larger than the length of the file. We define a function in the following, which will set the position to zero, if a negative value is applied. As there is no efficient way to check the length of a file and because it doesn't matter, if the position is greater than the length of the file, we will keep possible values greater than the length of a file.

```python
def relative_seek(fp, k):
    """ rel_seek moves the position of the file pointer k characte
rs to
    the left (k<0) or right (k>0)
    """
    position = fp.tell() + k
    if position < 0:
        position = 0
    fh.seek(position)


with open("small_text.txt") as fh:
    print(fh.tell())
    relative_seek(fh, 7)
    print(fh.tell())
    relative_seek(fh, -5)
    print(fh.tell())
    relative_seek(fh, -10)
    print(fh.tell())
```

```
0
7
2
0
```

You might have thought, when we wrote the function `relative_seek` why do we not use the second parameter of `seek`. After all the help file says "1 -- current stream position;". What the help file doesn't say is the fact that `seek` needs a file pointer opened with "br" (binary read), if the second parameter is set to 1 or 2. We show this in the next subchapter.

BINARY READ

So far we have only used the first parameter of `open`, i.e. the filename. The second parameter is optional and is set to "r" (read) by default. "r" means that the file is read in text mode. In text mode, if encoding

(another parameter of open) is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding.

The second parameter specifies the mode of access to the file or in other words the mode in which the file is opened. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding.

We will demonstrate this in the following example. To demonstrate the different effects we need a string which uses characters which are not included in standard ASCII. This is why we use a Turkish text, because it uses many special characters and Umlaute. the English translation means "See you, I'll come tomorrow.".

We will write a file with the Turkish text "Görüşürüz, yarın geleceğim.":

```
txt = "Görüşürüz, yarın geleceğim."
number_of_chars_written = open("see_you_tomorrow.txt", "w").writ
e(txt)
```

We will read in this files in text mode and binary mode to demonstrate the differences:

```
text = open("see_you_tomorrow.txt", "r").read()
print("text mode: ", text)
text_binary = open("see_you_tomorrow.txt", "rb").read()
print("binary mode: ", text_binary)
```

```
text mode:  Görüşürüz, yarın geleceğim.
binary mode:  b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz, yar\
xc4\xb1n gelece\xc4\x9fim.'
```

In binary mode, the characters which are not plain ASCII like "ö", "ü", "ş", "ğ" and "ı" are represented by more than one byte. In our case by two characters. 14 bytes are needed for "görüşürüz":

```
text_binary[:14]
```
Output: `b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'`

"ö" for example consists of the two bytes "\xc3" and "\xb6".

```
text[:9]
```
Output: `'Görüşürüz'`

There are two ways to turn a byte string into a string again:

```
t = text_binary.decode("utf-8")
```

```
print(t)
t2 = str(text_binary, "utf-8")
print(t2)
```

```
Görüşürüz, yarın geleceğim.
Görüşürüz, yarın geleceğim.
```

It is possible to use the values "1" and "2" for the second parameter of `seek`, if we open a file in binary format:

```
with open("see_you_tomorrow.txt", "rb") as fh:
    x = fh.read(14)
    print(x)
    # move 5 bytes to the right from the current position:
    fh.seek(5, 1)
    x = fh.read(3)
    print(x)
    print(str(x, "utf-8"))
    # let's move to the 8th byte from the right side of the byte s
tring:
    fh.seek(-8, 2)
    x = fh.read(5)
    print(x)
    print(str(x, "utf-8"))
```

```
b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'
b'\xc4\xb1n'
ın
b'ece\xc4\x9f'
eceğ
```

READ AND WRITE TO THE SAME FILE

In the following example we will open a file for reading and writing at the same time. If the file doesn't exist, it will be created. If you want to open an existing file for read and write, you should better use "r+", because this will not delete the content of the file.

```
fh = open('colours.txt', 'w+')
fh.write('The colour brown')

#Go to the 12th byte in the file, counting starts with 0
fh.seek(11)
print(fh.read(5))
print(fh.tell())
fh.seek(11)
```

```
fh.write('green')
fh.seek(0)
content = fh.read()
print(content)
```

```
brown
16
The colour green
```

"HOW TO GET INTO A PICKLE"

We don't really mean what the header says. On the contrary, we want to prevent any nasty situation, like losing the data, which your Python program has calculated. So, we will show you, how you can save your data in an easy way that you or better your program can reread them at a later date again. We are "pickling" the data, so that nothing gets lost.

Python offers a module for this purpose, which is called "pickle". With the algorithms of the pickle module we can serialize and de-serialize Python object structures. "Pickling" denotes the process which converts a Python object hierarchy into a byte stream, and "unpickling" on the other hand is the inverse operation, i.e. the byte stream is converted back into an object hierarchy. What we call pickling (and unpickling) is also known as "serialization" or "flattening" a data structure.

An object can be dumped with the dump method of the pickle module:

```
pickle.dump(obj, file[,protocol, *, fix_imports=True])
```

dump() writes a pickled representation of obj to the open file object file. The optional protocol argument tells the pickler to use the given protocol:

- Protocol version 0 is the original (before Python3) human-readable (ascii) protocol and is backwards compatible with previous versions of Python.
- Protocol version 1 is the old binary format which is also compatible with previous versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.
- Protocol version 3 was introduced with Python 3.0. It has explicit support for bytes and cannot be unpickled by Python 2.x pickle modules. It's the recommended protocol of Python 3.x.

The default protocol of Python3 is 3.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python3 names to the old module names used in Python2, so that the pickle data stream is readable with Python 2.

Objects which have been dumped to a file with pickle.dump can be reread into a program by using the method pickle.load(file). pickle.load recognizes automatically, which format had been used for writing the data. A simple example:

```
import pickle

cities = ["Paris", "Dijon", "Lyon", "Strasbourg"]
fh = open("data.pkl", "bw")
pickle.dump(cities, fh)
fh.close()
```

The file data.pkl can be read in again by Python in the same or another session or by a different program:

```
import pickle
f = open("data.pkl", "rb")
villes = pickle.load(f)
print(villes)
['Paris', 'Dijon', 'Lyon', 'Strasbourg']
```

Only the objects and not their names are saved. That's why we use the assignment to villes in the previous example, i.e. data = pickle.load(f).

In our previous example, we had pickled only one object, i.e. a list of French cities. But what about pickling multiple objects? The solution is easy: We pack the objects into another object, so we will only have to pickle one object again. We will pack two lists "programming_languages" and "python_dialects" into a list pickle_objects in the following example:

```
import pickle
fh = open("data.pkl","bw")
programming_languages = ["Python", "Perl", "C++", "Java", "Lisp"]
python_dialects = ["Jython", "IronPython", "CPython"]
pickle_object = (programming_languages, python_dialects)
pickle.dump(pickle_object,fh)
fh.close()
```

The pickled data from the previous example, - i.e. the data which we have written to the file data.pkl, - can be separated into two lists again, when we reread the data:

</pre> import pickle f = open("data.pkl","rb") languages, dialects) = pickle.load(f) print(languages, dialects) ['Python', 'Perl', 'C++', 'Java', 'Lisp'] ['Jython', 'IronPython', 'CPython'] </pre>

SHELVE MODULE

One drawback of the pickle module is that it is only capable of pickling one object at the time, which has to be unpickled in one go. Let's imagine this data object is a dictionary. It may be desirable that we don't have to save and load every time the whole dictionary, but save and load just a single value corresponding to just one key. The shelve module is the solution to this request. A "shelf" - as used in the shelve module - is a persistent, dictionary-like object. The difference with dbm databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the "pickle" module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys have to be strings.

The shelve module can be easily used. Actually, it is as easy as using a dictionary in Python. Before we can

use a shelf object, we have to import the module. After this, we have to open a shelve object with the shelve method open. The open method opens a special shelf file for reading and writing:

</pre> import shelve s = shelve.open("MyShelve")</pre>

If the file "MyShelve" already exists, the open method will try to open it. If it isn't a shelf file, - i.e. a file which has been created with the shelve module, - we will get an error message. If the file doesn't exist, it will be created.

We can use s like an ordinary dictionary, if we use strings as keys:

```
s["street"] = "Fleet Str"
s["city"] = "London"
for key in s:
    print(key)
```

A shelf object has to be closed with the close method:

```
 s.close()
```

We can use the previously created shelf file in another program or in an interactive Python session:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
import shelve
s = shelve.open("MyShelve")
s["street"]
'Fleet Str'
s["city"]
'London'
```

It is also possible to cast a shelf object into an "ordinary" dictionary with the dict function:

```
s
≤shelve.DbfilenameShelf object at 0xb7133dcc>
>>> dict(s)
{'city': 'London', 'street': 'Fleet Str'}
```

The following example uses more complex values for our shelf object:

```
import shelve
tele = shelve.open("MyPhoneBook")
tele["Mike"] = {"first":"Mike", "last":"Miller", "phone":"4689"}
tele["Steve"] = {"first":"Stephan", "last":"Burns", "phone":"8745"}
tele["Eve"] = {"first":"Eve", "last":"Naomi", "phone":"9069"}
```

```
tele["Eve"]["phone"]
'9069'
```

The data is persistent!

To demonstrate this once more, we reopen our MyPhoneBook:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
    import shelve
    tele = shelve.open("MyPhoneBook")
    tele["Steve"]["phone"]
'8745'
```

EXERCISES

Write a function which reads in a text from file and returns a list of the paragraphs. You may use one of the following books:

- Virginia Woolf: To the Lighthouse
- Samuel Butler: The Way of all Flash
- Herman Melville: Moby Dick
- David Herbert Lawrence: Sons and Lovers
- Daniel Defoe: The Life and Adventures of Robinson Crusoe
- James Joyce: Ulysses

Save the following text containing city names and times as "cities_and_times.txt".

```
Chicago Sun 01:52
Columbus Sun 02:52
Riyadh   Sun 10:52
Copenhagen  Sun 08:52
Kuwait City Sun 10:52
Rome     Sun 08:52
Dallas   Sun 01:52
Salt Lake City  Sun 01:52
```

```
San Francisco Sun 00:52
Amsterdam Sun 08:52
Denver Sun 01:52
San Salvador Sun 01:52
Detroit Sun 02:52
Las Vegas Sun 00:52
Santiago    Sun 04:52
Anchorage Sat 23:52
Ankara Sun 10:52
Lisbon  Sun 07:52
São Paulo   Sun 05:52
Dubai   Sun 11:52
London  Sun 07:52
Seattle Sun 00:52
Dublin  Sun 07:52
Los Angeles Sun 00:52
Athens  Sun 09:52
Edmonton Sun 01:52
Madrid  Sun 08:52
Shanghai Sun 15:52
Atlanta Sun 02:52
Frankfurt   Sun 08:52
Singapore Sun 15:52
Auckland Sun 20:52
Halifax Sun 03:52
Melbourne Sun 18:52
Stockholm   Sun 08:52
Barcelona   Sun 08:52
Miami   Sun 02:52
Minneapolis Sun 01:52
Sydney Sun 18:52
Beirut  Sun 09:52
Helsinki    Sun 09:52
Montreal    Sun 02:52
Berlin  Sun 08:52
Houston Sun 01:52
Moscow  Sun 10:52
Indianapolis    Sun 02:52
Boston  Sun 02:52
Tokyo   Sun 16:52
Brasilia Sun 05:52
Istanbul Sun 10:52
Toronto Sun 02:52
Vancouver   Sun 00:52
Brussels    Sun 08:52
Jerusalem   Sun 09:52
New Orleans Sun 01:52
Vienna  Sun 08:52
```

```
Bucharest    Sun 09:52
Johannesburg    Sun 09:52
New York    Sun 02:52
Warsaw  Sun 08:52
Budapest    Sun 08:52
Oslo    Sun 08:52
Washington DC    Sun 02:52
Ottawa  Sun 02:52
Winnipeg    Sun 01:52
Cairo   Sun 09:52
Paris   Sun 08:52
Calgary Sun 01:52
Kathmandu    Sun 13:37
Philadelphia    Sun 02:52
Zurich  Sun 08:52
Cape Town    Sun 09:52
Phoenix Sun 00:52
Prague  Sun 08:52
Casablanca    Sun 07:52
Reykjavik    Sun 07:52
```

Each line contains the name of the city, followed by the name of the day ("Sun") and the time in the form hh:mm. Read in the file and create an alphabetically ordered list of the form

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)), ('A
nkara', 'Sun', (10, 52)), ('Athens', 'Sun', (9, 52)), ('Atlanta',
'Sun', (2, 52)), ('Auckland', 'Sun', (20, 52)), ('Barcelona', 'Su
n', (8, 52)), ('Beirut', 'Sun', (9, 52)),

...

  ('Toronto', 'Sun', (2, 52)), ('Vancouver', 'Sun', (0, 52)), ('Vien
na', 'Sun', (8, 52)), ('Warsaw', 'Sun', (8, 52)), ('Washington D
C', 'Sun', (2, 52)), ('Winnipeg', 'Sun', (1, 52)), ('Zurich', 'Su
n', (8, 52))]
```

Finally, the list should be dumped for later usage with the pickle module. We will use this list in our chapter on Numpy dtype.

SOLUTIONS

SOLUTION 1

```python
def text2paragraphs(filename, min_size=1):
```

```
    """ A text contained in the file 'filename' will be read
    and chopped into paragraphs.
    Paragraphs with a string length less than min_size will be ign
ored.
    A list of paragraph strings will be returned"""

    txt = open(filename).read()
    paragraphs = [para for para in txt.split("\n\n") if len(para)
> min_size]
    return paragraphs

paragraphs = text2paragraphs("books/to_the_lighthouse_woolf.txt",
min_size=100)

for i in range(10, 14):
    print(paragraphs[i])
```

"I should think there would be no one to talk to in Manchester," she
replied at random. Mr. Fortescue had been observing her for a moment or
two, as novelists are inclined to observe, and at this remark he smiled,
and made it the text for a little further speculation.
"In spite of a slight tendency to exaggeration, Katharine decidedly
hits the mark," he said, and lying back in his chair, with his opaque
contemplative eyes fixed on the ceiling, and the tips of his fingers
pressed together, he depicted, first the horrors of the streets of
Manchester, and then the bare, immense moors on the outskirts of the
town, and then the scrubby little house in which the girl would live,
and then the professors and the miserable young students devoted to the
more strenuous works of our younger dramatists, who would visit her,
and how her appearance would change by degrees, and how she would fly to
London, and how Katharine would have to lead her about, as one leads an
eager dog on a chain, past rows of clamorous butchers' shops, poor dear
creature.
"Oh, Mr. Fortescue," exclaimed Mrs. Hilbery, as he finished, "I had just
written to say how I envied her! I was thinking of the big gardens and
the dear old ladies in mittens, who read nothing but the "Spectator,"
 and snuff the candles. Have they ALL disappeared? I told her she would
find the nice things of London without the horrid streets that depress
one so."
"There is the University," said the thin gentleman, who had previously
insisted upon the existence of people knowing Persian.

```
import pickle

lines = open("cities_and_times.txt").readlines()
lines.sort()

cities = []
for line  in lines:
    *city, day, time = line.split()
     hours, minutes = time.split(":")
      cities.append((" ".join(city), day, (int(hours), int(minute
s)) ))

fh = open("cities_and_times.pkl", "bw")
pickle.dump(cities, fh)
```

City names can consist of multiple words like "Salt Lake City". That is why we have to use the asterisk in the line, in which we split a line. So city will be a list with the words of the city, e.g. ["Salt", "Lake", "City"]. " ".join(city) turns such a list into a "proper" string with the city name, i.e. in our example "Salt Lake City".

MODULAR PROGRAMMING AND MODULES

MODULAR PROGRAMMING

Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.

There is hardly any product nowadays, which doesn't heavily rely on modularisation, like cars and mobile phones. Computers belong to those products which are modularised to the utmost. So, what's a must for the hardware is an unavoidable necessity for the software running on the computers.

If you want to develop programs which are readable, reliable and maintainable without too much effort, you have to use some kind of modular software design. Especially if your application has a certain size. There exists a variety of concepts to design software in modular form. Modular programming is a software design technique to split your code into separate parts. These parts are called modules. The focus for this separation should be to have modules with no or just few dependencies upon other modules. In other words: Minimization of dependencies is the goal. When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

IMPORTING MODULES

So far we haven't explained what a Python module is. In a nutshell: every file, which has the file extension .py and consists of proper Python code, can be seen or is a module! There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:

In [ ]:

```python
import math
```

The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:

```
math.pi
```
Output: `3.141592653589793`

```
math.sin(math.pi/2)
```
Output: `1.0`

```
math.cos(math.pi/2)
```
Output: `6.123233995736766e-17`

```
math.cos(math.pi)
```
Output: `-1.0`

It's possible to import more than one module in one import statement. In this case the module names are separated by commas:

In [ ]:
```
import math, random
```

import statements can be positioned anywhere in the program, but it's good style to place them directly at the beginning of a program.

If only certain objects of a module are needed, we can import only those:

In [ ]:
```
from math import sin, pi
```

The other objects, e.g. cos, are not available after this import. We are capable of accessing sin and pi directly, i.e. without prefixing them with "math." Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:

```
from math import *
sin(3.01) + tan(cos(2.1)) + e
```
Output: `2.2968833711382604`

```
e
```
Output: `2.718281828459045`

It's not recommended to use the asterisk notation in an import statement, except when working in the interactive Python shell. One reason is that the origin of a name can be quite obscure, because it can't be seen

from which module it might have been imported. We will demonstrate another serious complication in the following example:

```python
from numpy import *
from math import *
print(sin(3))
```

```
0.1411200080598672
```

```python
sin(3)
```

Output: 0.1411200080598672

Let's modify the previous example slightly by changing the order of the imports:

```python
from math import *
from numpy import *
print(sin(3))
```

```
0.1411200080598672
```

```python
sin(3)
```

Output: 0.1411200080598672

People use the asterisk notation, because it is so convenient. It means avoiding a lot of tedious typing. Another way to shrink the typing effort lies in renaming a namespace. A good example for this is the numpy module. You will hardly find an example or a tutorial, in which they will import this module with the statement.

```python
import numpy
```

It's like an unwritten law to import it with

```python
import numpy as np
```

Now you can prefix all the objects of numpy with "np." instead of "numpy.":

```python
import numpy as np
np.diag([3, 11, 7, 9])
```

Output:
```
array([[ 3,  0,  0,  0],
       [ 0, 11,  0,  0],
       [ 0,  0,  7,  0],
       [ 0,  0,  0,  9]])
```

DESIGNING AND WRITING MODULES

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done, we just save the following code in the file fibonacci.py:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
def ifib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

The newly created module "fibonacci" is ready for use now. We can import this module like any other module in a program or script. We will demonstrate this in the following interactive Python shell:

```python
import fibonacci
fibonacci.fib(7)
```
Output: 13

```python
fibonacci.fib(20)
```
Output: 6765

```python
fibonacci.ifib(42)
```
Output: 267914296

```python
fibonacci.ifib(1000)
```
Output:
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875

Don't try to call the recursive version of the Fibonacci function with large arguments like we did with the iterative version. A value like 42 is already too large. You will have to wait for a long time!

As you can easily imagine: It's a pain if you have to use those functions often in your program and you always have to type in the fully qualified name, i.e. fibonacci.fib(7). One solution consists in assigning a local name to a module function to get a shorter name:

```
fib = fibonacci.ifib
fib(10)
```

Output: 55

But it's better, if you import the necessary functions directly into your module, as we will demonstrate further down in this chapter.

MORE ON MODULES

Usually, modules contain functions or classes, but there can be "plain" statements in them as well. These statements can be used to initialize the module. They are only executed when the module is imported.

Let's look at a module, which only consists of just one statement:

```
print("The module is imported now!")
```

```
The module is imported now!
```

We save with the name "one_time.py" and import it two times in an interactive session:

```
import one_time
```

```
The module is imported now!
```

We can see that it was only imported once. Each module can only be imported once per interpreter session or in a program or script. If you change a module and if you want to reload it, you must restart the interpreter again. In Python 2.x, it was possible to reimport the module by using the built-in reload, i.e.reload(modulename):

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
import one_time
The module is imported now!
reload(one_time)
The module is imported now!
```

This is not possible anymore in Python 3.x. You will cause the following error:

```
import one_time
```

```
reload(one_time)
```

```
---------------------------------------------------------------------------
NameError                                    Traceback (most recent call last)
<ipython-input-7-102e1bec2702> in <module>
----> 1 reload(one_time)

NameError: name 'reload' is not defined
```

Since Python 3.0 the reload built-in function has been moved into the imp standard library module. So it's still possible to reload files as before, but the functionality has to be imported. You have to execute an "import imp" and use imp.reload(my_module). Alternatively, you can use "imp import reload" and use reload(my_module).

Example with reloading the Python3 way:

```
$ python3
Python 3.1.2 (r312:79147, Sep 27 2010, 09:57:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more informati
on.
from imp import reload
import one_time
The module is imported now!
reload(one_time)
The module is imported now!
```

Since version 3.4 you should use the "importlib" module, because imp.reload is marked as deprecated:

```
from importlib import reload
import one_time
The module is imported now!
reload(one_time)
The module is imported now!
```

A module has a private symbol table, which is used as the global symbol table by all functions defined in the module. This is a way to prevent that a global variable of a module accidentally clashes with a user's global variable with the same name. Global variables of a module can be accessed with the same notation as functions, i.e. modname.name A module can import other modules. It is customary to place all import statements at the beginning of a module or a script.

IMPORTING NAMES FROM A MODULE DIRECTLY

Names from a module can directly be imported into the importing module's symbol table:

```python
from fibonacci import fib, ifib
ifib(500)
```

Output: 13942322456169788013972438287040728395007025658769730726410896
29483255716228632906915576588876222521294125

This does not introduce the module name from which the imports are taken in the local symbol table. It's possible but not recommended to import all names defined in a module, except those beginning with an underscore "_":

In [ ]:
```python
from fibonacci import *
fib(500)
```

This shouldn't be done in scripts but it's possible to use it in interactive sessions to save typing.

EXECUTING MODULES AS SCRIPTS

Essentially a Python module is a script, so it can be run as a script:

```
python fibo.py
```

The module which has been started as a script will be executed as if it had been imported, but with one exception: The system variable **name** is set to "**main**". So it's possible to program different behaviour into a module for the two cases. With the following conditional statement the file can be used as a module or as a script, but only if it is run as a script the method fib will be started with a command line argument:

```python
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

If it is run as a script, we get the following output:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If it is imported, the code in the if block will not be executed:

```
import fibo
```

RENAMING A NAMESPACE

While importing a module, the name of the namespace can be changed:

```python
import math as mathematics
print(mathematics.cos(mathematics.pi))
```

```
-1.0
```

After this import there exists a namespace mathematics but no namespace math. It's possible to import just a few methods from a module:

```python
from math import pi, pow as power, sin as sinus
power(2,3)
```
Output: `8.0`

```python
sinus(pi)
```
Output: `1.2246467991473532e-16`

KINDS OF MODULES

There are different kind of modules:

- Those written in Python
  They have the suffix: .py
- Dynamically linked C modules
  Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter
  It's possible to get a complete list of these modules:

```python
import sys
print(sys.builtin_module_names)
```

```
('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn',
'_codecs_hk', '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_cod
ecs_tw', '_collections', '_contextvars', '_csv', '_datetime', '_fu
nctools', '_heapq', '_imp', '_io', '_json', '_locale', '_lsprof',
'_md5', '_multibytecodec', '_opcode', '_operator', '_pickle', '_ra
ndom', '_sha1', '_sha256', '_sha3', '_sha512', '_signal', '_sre',
'_stat', '_string', '_struct', '_symtable', '_thread', '_tracemall
oc', '_warnings', '_weakref', '_winapi', 'array', 'atexit', 'audio
op', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'g
c', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parse
r', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')
```

An error message is returned for Built-in-Modules.

MODULE SEARCH PATH

If you import a module, let's say "import xyz", the interpreter searches for this module in the following locations and in the order given:

```
The directory of the top-level file, i.e. the file being executed.
The directories of PYTHONPATH, if this global environment variable
of your operating system is set.
standard installation path Linux/Unix e.g. in /usr/lib/python3.5.
```

It's possible to find out where a module is located after it has been imported:

```
import numpy
numpy.file
'/usr/lib/python3/dist-packages/numpy/init.py'

import random
random.file
'/usr/lib/python3.5/random.py'

</pre>
The file attribute doesn't always exist. This is the case with modu
les which are statically linked C libraries.
```

```
import math
math.__file__
```

```
-------------------------------------------------------------------
AttributeError                               Traceback (most recent call last)
<ipython-input-4-bb98ec32d2a8> in <module>
      1 import math
----> 2 math.__file__

AttributeError: module 'math' has no attribute '__file__'
```

CONTENT OF A MODULE

With the built-in function dir() and the name of the module as an argument, you can list all valid attributes and methods for that module.

```python
import math
dir(math)
```

Output:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
```

```
        'radians',
        'remainder',
        'sin',
        'sinh',
        'sqrt',
        'tan',
        'tanh',
        'tau',
        'trunc']
```

Calling dir() without an argument, a list with the names in the current local scope is returned:

```
import math
cities = ["New York", "Toronto", "Berlin", "Washington", "Amsterda
m", "Hamburg"]
dir()
```

Output:
```
['In',
 'Out',
 '_',
 '_1',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'builtins',
 'cities',
 'exit',
 'get_ipython',
 'math',
 'quit']
```

It's possible to get a list of the Built-in functions, exceptions, and other objects by importing the builtins module:

```
import builtins
dir(builtins)
```

Output: ['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',

```
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
```

```
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
```

```
        'range',
        'repr',
        'reversed',
        'round',
        'set',
        'setattr',
        'slice',
        'sorted',
        'staticmethod',
        'str',
        'sum',
        'super',
        'tuple',
        'type',
        'vars',
        'zip']
```

PACKAGES

If you have created a lot of modules at some point in time, you may loose the overview about them. You may have dozens or hundreds of modules and they can be categorized into different categories. It is similar to the situation in a file system: Instead of having all files in just one directory, you put them into different ones, being organized according to the topics of the files. We will show in the next chapter of our Python tutorial how to organize modules into packages.

REGULAR EXPRESSIONS

The aim of this chapter of our Python tutorial is to present a detailed and descriptive introduction into regular expressions. This introduction will explain the theoretical aspects of regular expressions and will show you how to use them in Python scripts.

The term "regular expression", sometimes also called regex or regexp, has originated in theoretical computer science. In theoretical computer science, they are used to define a language family with certain characteristics, the so-called regular languages. A finite state machine (FSM), which accepts language defined by a regular expression, exists for every regular expression. You can find an implementation of a Finite State Machine in Python on our website.

Regular Expressions are used in programming languages to filter texts or textstrings. It's possible to check, if a text or a string matches a regular expression. A great thing about regular expressions: The syntax of regular expressions is the same for all programming and script languages, e.g. Python, Perl, Java, SED, AWK and even X#.

The first programs which had incorporated the capability to use regular expressions were the Unix tools ed (editor), the stream editor sed and the filter grep.

There is another mechanism in operating systems, which shouldn't be mistaken for regular expressions. Wildcards, also known as globbing, look very similar in their syntax to regular expressions. However, the semantics differ considerably. Globbing is known from many command line shells, like the Bourne shell, the Bash shell or even DOS. In Bash e.g. the command "ls *.txt" lists all files (or even directories) ending with the suffix .txt; in regular expression notation ".txt" wouldn't make sense, it would have to be written as ".*.txt"
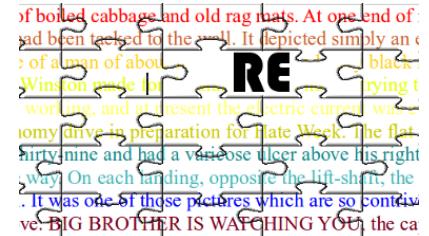
INTRODUCTION

When we introduced the sequential data types, we got to know the "in" operator. We check in the following example, if the string "easily" is a substring of the string "Regular expressions easily explained!":

```
s = "Regular expressions easily explained!"
"easily" in s
```
Output: True

We show step by step with the following diagrams how this matching is performed: We check if the string sub = "abc"
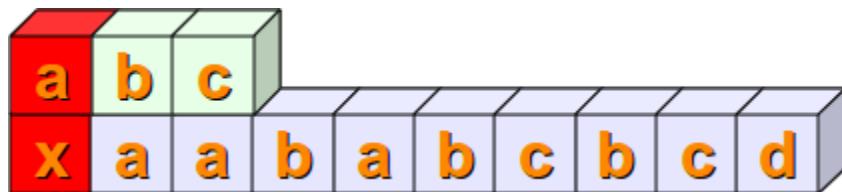
is contained in the string s = "xaababcbcd"
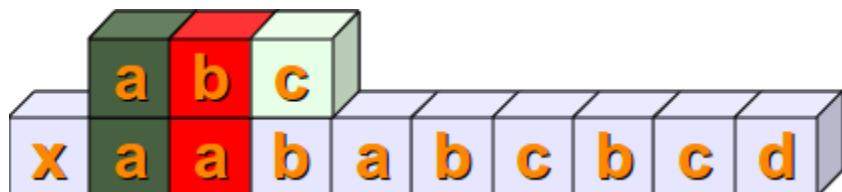
By the way, the string sub = "abc" can be seen as a regular expression, just a very simple one.
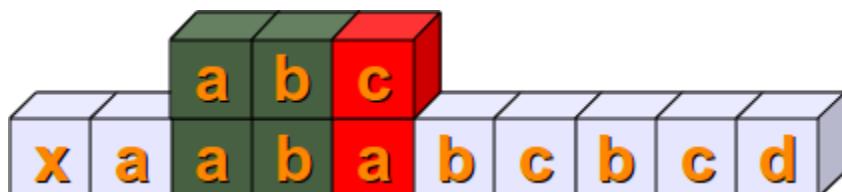
In the first place, we check, if the first positions of the two string match, i.e. s[0] == sub[0]. This is not satisfied in our example. We mark this fact by the colour red:
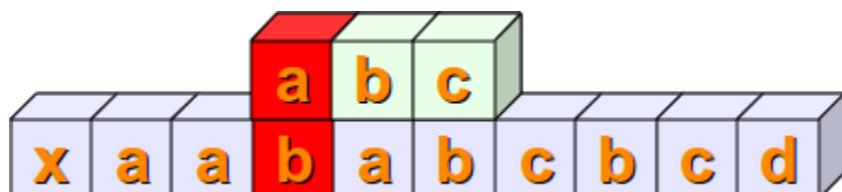
Then we check, if s[1:4] == sub. In other words, we have to check at first, if sub[0] is equal to s[1]. This is true and we mark it with the colour green. Then, we have to compare the next positions. s[2] is not equal to sub[1], so we don't have to proceed further with the next position of sub and s:
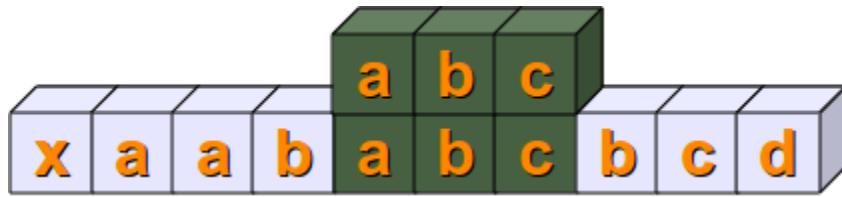
Now we have to check if s[2:5] and sub are equal. The first two positions are equal but not the third:

The following steps should be clear without any explanations:

Finally, we have a complete match with s[4:7] == sub :

A SIMPLE REGULAR EXPRESSION

As we have already mentioned in the previous section, we can see the variable "sub" from the introduction as a very simple regular expression. If you want to use regular expressions in Python, you have to import the re module, which provides methods and functions to deal with regular expressions.

REPRESENTING REGULAR EXPRESSIONS IN PYTHON

From other languages you might be used to representing regular expressions within Slashes "/", e.g. that's the way Perl, SED or AWK deals with them. In Python there is no special notation. Regular expressions are represented as normal strings.

But this convenience brings along a small problem: The backslash is a special character used in regular expressions, but is also used as an escape character in strings. This implies that Python would first evaluate every backslash of a string and after this - without the necessary backslashes - it would be used as a regular expression. One way to prevent this could be writing every backslash as "\" and this way keep it for the evaluation of the regular expression. This can cause extremely clumsy expressions. E.g. a backslash in a regular expression has to be written as a double backslash, because the backslash functions as an escape character in regular expressions. Therefore, it has to be quoted. The same is valid for Python strings. The backslash has to be quoted by a backslash. So, a regular expression to match the Windows path "C:\programs" corresponds to a string in regular expression notation with four backslashes, i.e. "C:\\programs".

The best way to overcome this problem would be marking regular expressions as raw strings. The solution to our Windows path example looks like this as a raw string:

```
r"C:\\programs"
```

Let's look at another example, which might be quite disturbing for people who are used to wildcards:

```
r"^a.*\.html$"
```

The regular expression of our previous example matches all file names (strings) which start with an "a" and end with ".html". We will the structure of the example above in detail explain in the following sections.

SYNTAX OF REGULAR EXPRESSION
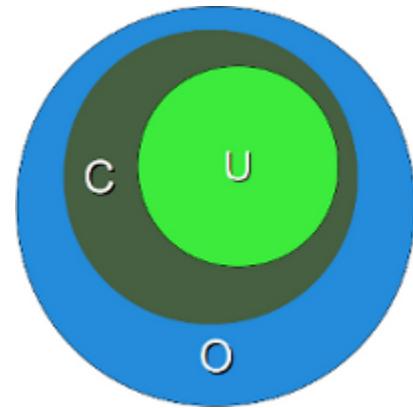
```
r"cat"
```

is a regular expression, though a very simple one without any metacharacters. Our RE

```
r"cat"
```

matches, for example, the following string: "A cat and a rat can't be friends."

Interestingly, the previous example shows already a "favourite" example for a mistake, frequently made not only by beginners and novices but also by advanced users of regular expressions. The idea of this example is to match strings containing the word "cat". We are successful at this, but unfortunately we are matching a lot of other words as well. If we match "cats" in a string that might be still okay, but what about all those words containing this character sequence "cat"? We match words like "education", "communicate", "falsification", "ramifications", "cattle" and many more. This is a case of "over matching", i.e. we receive positive results, which are wrong according to the problem we want to solve.

We have illustrated this problem in the diagram above. The dark green Circle C corresponds to the set of "objects" we want to recognize. But instead we match all the elements of the set O (blue circle). C is a subset of O. The set U (light green circle) in this diagram is a subset of C. U is a case of "under matching", i.e. if the regular expression is not matching all the intended strings. If we try to fix the previous RE, so that it doesn't create over matching, we might try the expression

```
r" cat "
```

. These blanks prevent the matching of the above mentioned words like "education", "falsification" and "ramification", but we fall prey to another mistake. What about the string "The cat, called Oscar, climbed on the roof."? The problem is that we don't expect a comma but only a blank surrounding the word "cat".

Before we go on with the description of the syntax of regular expressions, we want to explain how to use them in Python:

```python
import re
x = re.search("cat", "A cat and a rat can't be friends.")
print(x)
```

```
<re.Match object; span=(2, 5), match='cat'>
```

```python
x = re.search("cow", "A cat and a rat can't be friends.")
print(x)
```

```
None
```

In the previous example we had to import the module re to be able to work with regular expressions. Then we used the method search from the re module. This is most probably the most important and the most often used method of this module. re.search(expr,s) checks a string s for an occurrence of a substring which matches the regular expression expr. The first substring (from left), which satisfies this condition will be returned. If a match has been possible, we get a so-called match object as a result, otherwise the value will be None. This method is already enough to use regular expressions in a basic way in Python programs. We can use it in conditional statements: If a regular expression matches, we get an SRE object returned, which is taken as a True value, and None, which is the return value if it doesn't match, is taken as False:

```python
if re.search("cat", "A cat and a rat can't be friends."):
    print("Some kind of cat has been found :-)")
else:
    print("No cat has been found :-)")
```

```
Some kind of cat has been found :-)
```

```python
if re.search("cow", "A cat and a rat can't be friends."):
    print("Cats and Rats and a cow.")
else:
    print("No cow around.")
```

```
No cow around.
```
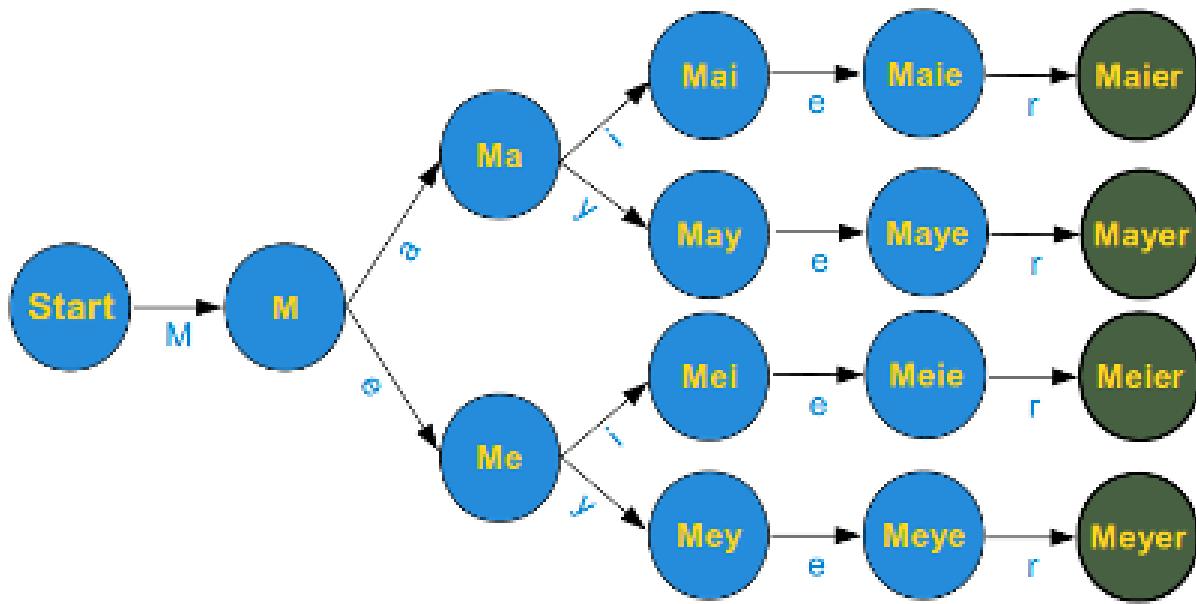
Let's assume that we have not been interested in the previous example to recognize the word cat, but all three letter words, which end with "at". The syntax of regular expressions supplies a metacharacter ".", which is used like a placeholder for "any character". The regular expression of our example can be written like this: r" .at " This RE matches three letter words, isolated by blanks, which end in "at". Now we get words like "rat", "cat", "bat", "eat", "sat" and many others.

But what if the text contains "words" like "@at" or "3at"? These words match as well, meaning we have caused over matching again. We will learn a solution in the following section.

Square brackets, "[" and "]", are used to include a character class. [xyz] means e.g. either an "x", an "y" or a "z". Let's look at a more practical example:

```
r"M[ae][iy]er"
```

This is a regular expression, which matches a surname which is quite common in German. A name with the same pronunciation and four different spellings: Maier, Mayer, Meier, Meyer A finite state automata to recognize this expression can be build like this:

The graph of the finite state machine (FSM) is simplified to keep the design easy. There should be an arrow in the start node pointing back on its own, i.e. if a character other than an upper case "M" has been processed, the machine should stay in the start condition. Furthermore, there should be an arrow pointing back from all nodes except the final nodes (the green ones) to the start node, unless the expected letter has been processed. E.g. if the machine is in state Ma, after having processed a "M" and an "a", the machine has to go back to state "Start", if any character except "i" or "y" can be read. Those who have problems with this FSM, shouldn't worry, since it is not a prerequisite for the rest of the chapter.

Instead of a choice between two characters, we often need a choice between larger character classes. We might need e.g. a class of letters between "a" and "e" or between "0" and "5". To manage such character classes, the syntax of regular expressions supplies a metacharacter "-". [a-e] a simplified writing for [abcde] or [0-5] denotes [012345].

The advantage is obvious and even more impressive, if we have to coin expressions like "any uppercase letter" into regular expressions. So instead of [ABCDEFGHIJKLMNOPQRSTUVWXYZ] we can write [A-Z]. If this is not convincing: Write an expression for the character class "any lower case or uppercase letter" [A-Za-z]

There is something more about the dash, we used to mark the begin and the end of a character class. The dash has only a special meaning if it is used within square brackets and in this case only if it isn't positioned directly after an opening or immediately in front of a closing bracket. So the expression [-az] is only the choice between the three characters "-", "a" and "z", but no other characters. The same is true for [az-].

Exercise: What character class is described by [-a-z]?

Answer The character "-" and all the characters "a", "b", "c" all the way up to "z".

The only other special character inside square brackets (character class choice) is the caret "^". If it is used directly after an opening sqare bracket, it negates the choice. [^0-9] denotes the choice "any character but a digit". The position of the caret within the square brackets is crucial. If it is not positioned as the first character following the opening square bracket, it has no special meaning. [^abc] means anything but an "a", "b" or "c"

[a^bc] means an "a", "b", "c" or a "^"

Before we go on with our introduction into regular expressions, we want to insert a practical exercise with Python. We have a phone list of the Simpsons, yes, the famous Simpsons from the American animated TV series. There are some people with the surname Neu. We are looking for a Neu, but we don't know the first name, we just know that it starts with a J. Let's write a Python script, which finds all the lines of the phone book, which contain a person with the described surname and a first name starting with J. If you don't know how to read and work with files, you should work through our chapter File Management. So here is our example script:

```python
import re

fh = open("simpsons_phone_book.txt")
for line in fh:
    if re.search(r"J.*Neu",line):
        print(line.rstrip())
fh.close()
```

```
Jack Neu 555-7666
Jeb Neu 555-5543
Jennifer Neu 555-3652
```

Instead of downloading simpsons_phone_book.txt, we can also use the file directly from the website by using urlopen from the module urllib.request:

```python
import re

from urllib.request import urlopen
with urlopen('https://www.python-course.eu/simpsons_phone_book.txt') as fh:
    for line in fh:
        # line is a byte string so we transform it to utf-8:
        line = line.decode('utf-8').rstrip()
        if re.search(r"J.*Neu",line):
            print(line)
```

```
Jack Neu 555-7666
Jeb Neu 555-5543
Jennifer Neu 555-3652
```

You might have realized that it can be quite cumbersome to construe certain character classes. A good example is the character class, which describes a valid word character. These are all lower case and uppercase characters plus all the digits and the underscore, corresponding to the following regular expression: r"[a-zA-Z0-9_]"

```
The special sequences consist of "\\" and a character from the foll
owing list:
\d  Matches any decimal digit; equivalent to the set [0-9].
\D  The complement of \d. It matches any non-digit character; equiv
alent to the set [^0-9].
\s  Matches any whitespace character; equivalent to [ \t\n\r\f\v].
\S  The complement of \s. It matches any non-whitespace character;
equiv. to [^ \t\n\r\f\v].
\w  Matches any alphanumeric character; equivalent to [a-zA-Z
0-9_]. With LOCALE, it will match the set [a-zA-Z0-9_] plus charact
ers defined as letters for the current locale.
\W  Matches the complement of \w.
\b  Matches the empty string, but only at the start or end of a wor
d.
\B  Matches the empty string, but not at the start or end of a wor
d.
\\  Matches a literal backslash.
```

The \b and \B of the previous overview of special sequences, is often not properly understood or even misunderstood especially by novices. While the other sequences match characters, - e.g. \w matches characters like "a", "b", "m", "3" and so on, - \b and \B don't match a character. They match empty strings depending on their neighbourhood, i.e. what kind of a character the predecessor and the successor is. So \b matches any empty string between a \W and a \w character and also between a \w and a \W character. \B is the complement, i.e empty strings between \W and \W or empty strings between \w and \w. We illustrate this in the following example:

We will get to know further "virtual" matching characters, i.e. the caret (^), which is used to mark the beginning of a string, and the dollar sign ($), which is used to mark the end of a string, respectively. \A and \Z, which can also be found in our previous diagram, are very seldom used as alternatives to the caret and the dollar sign.

MATCHING BEGINNING AND END

As we have previously carried out in this introduction, the expression r"M[ae][iy]er" is capable of matching various spellings of the name Mayer and the name can be anywhere in the string:

```
import re
line = "He is a German called Mayer."
if re.search(r"M[ae][iy]er", line):
    print("I found one!")
```

```
I found one!
```

But what if we want to match a regular expression at the beginning of a string and only at the beginning?

The re module of Python provides two functions to match regular expressions. We have met already one of them, i.e. search(). The other has in our opinion a misleading name: match() Misleading, because match(re_str, s) checks for a match of re_str merely at the beginning of the string. But anyway, match() is the solution to our question, as we can see in the following example:

```
import re
s1 = "Mayer is a very common Name"
s2 = "He is called Meyer but he isn't German."
print(re.search(r"M[ae][iy]er", s1))
print(re.search(r"M[ae][iy]er", s2))
 # matches because it starts with Mayer
print(re.match(r"M[ae][iy]er", s1))
# doesn't match because it doesn't start with Meyer or Meyer, Meie
r and so on:
print(re.match(r"M[ae][iy]er", s2))
```

```
<re.Match object; span=(0, 5), match='Mayer'>
<re.Match object; span=(13, 18), match='Meyer'>
<re.Match object; span=(0, 5), match='Mayer'>
None
```

So, this is a way to match the start of a string, but it's a Python specific method, i.e. it can't be used in other languages like Perl, AWK and so on. There is a general solution which is a standard for regular expressions:

The caret '^' matches the start of the string, and in MULTILINE (will be explained further down) mode also matches immediately after each newline, which the Python method match() doesn't do. The caret has to be the first character of a regular expression:

```
import re
s1 = "Mayer is a very common Name"
s2 = "He is called Meyer but he isn't German."
print(re.search(r"^M[ae][iy]er", s1))
print(re.search(r"^M[ae][iy]er", s2))
```

```
<re.Match object; span=(0, 5), match='Mayer'>
None
```

But what happens if we concatenate the two strings s1 and s2 in the following way?

```
s = s2 + "\n" + s1
```

Now the string doesn't start with a Maier of any kind, but the name follows a newline character:

```
s = s2 + "\n" + s1
print(re.search(r"^M[ae][iy]er", s))
```

```
None
```

The name hasn't been found, because only the beginning of the string is checked. It changes, if we use the multiline mode, which can be activated by adding the following parameters to search:

```
print(re.search(r"^M[ae][iy]er", s, re.MULTILINE))
print(re.search(r"^M[ae][iy]er", s, re.M))
print(re.match(r"^M[ae][iy]er", s, re.M))
```

```
<re.Match object; span=(40, 45), match='Mayer'>
<re.Match object; span=(40, 45), match='Mayer'>
None
```

The previous example also shows that the multiline mode doesn't affect the match method. match() never checks anything but the beginning of the string for a match.

We have learnt how to match the beginning of a string. What about the end? Of course that's possible to. The dollar sign *""isusedasametacharacterforthispurpose."* matches the end of a string or just before the newline at the end of the string. If in MULTILINE mode, it also matches before a newline. We demonstrate the usage of the "$" character in the following example:

```
print(re.search(r"Python\.$","I like Python."))
print(re.search(r"Python\.$","I like Python and Perl."))
print(re.search(r"Python\.$","I like Python.\nSome prefer Java or
Perl."))
print(re.search(r"Python\.$","I like Python.\nSome prefer Java or
Perl.", re.M))
```

```
<re.Match object; span=(7, 14), match='Python.'>
None
None
<re.Match object; span=(7, 14), match='Python.'>
```

If you thought that our collection of Mayer names was complete, you were wrong. There are other ones all over the world, e.g. London and Paris, who dropped their "e". So we have four more names ["Mayr", "Meyr", "Meir", "Mair"] plus our old set ["Mayer", "Meyer", "Meier", "Maier"].

If we try to figure out a fitting regular expression, we realize that we miss something. A way to tell the computer "this "e" may or may not occur". A question mark is used as a notation for this. A question mark declares that the preceding character or expression is optional.

The final Mayer-Recognizer looks now like this:

```
r"M[ae][iy]e?r"
```

A subexpression is grouped by round brackets and a question mark following such a group means that this group may or may not exist. With the following expression we can match dates like "Feb 2011" or February 2011":

```
r"Feb(ruary)? 2011"
```

If you just use what we have introduced so far, you will still need a lot of things, above all some way of repeating characters or regular expressions. For this purpose, quantifiers are used. We have encountered one in the previous paragraph, i.e. the question mark.

A quantifier after a token, which can be a single character or group in brackets, specifies how often that preceding element is allowed to occur. The most common quantifiers are

- the question mark ?
- the asterisk or star character *, which is derived from the Kleene star
- and the plus sign +, derived from the Kleene cross

We have already previously used one of these quantifiers without explaining it, i.e. the asterisk. A star following a character or a subexpression group means that this expression or character may be repeated arbitrarily, even zero times.

```
r"[0-9]*"
```

The above expression matches any sequence of digits, even the empty string. r".*" matches any sequence of characters and the empty string.

Exercise: Write a regular expression which matches strings which starts with a sequence of digits - at least one digit - followed by a blank.

Solution:

```
r"^[0-9][0-9]* "
```

So, you used the plus character "+". That's fine, but in this case you have either cheated by going ahead in the text or you know already more about regular expressions than we have covered in our course :-)

Now that we mentioned it: The plus operator is very convenient to solve the previous exercise. The plus operator is very similar to the star operator, except that the character or subexpression followed by a "+" sign has to be repeated at least one time. Here follows the solution to our exercise with the plus quantifier:

Solution with the plus quantifier:

```
r"^[0-9]+ "
```

If you work with this arsenal of operators for a while, you will inevitably miss the possibility to repeat expressions for an exact number of times at some point. Let's assume you want to recognize the last lines of addresses on envelopes in Switzerland. These lines usually contain a four digits long post code followed by a blank and a city name. Using + or * are too unspecific for our purpose and the following expression seems to be too clumsy:

```
r"^[0-9][0-9][0-9][0-9] [A-Za-z]+"
```

Fortunately, there is an alternative available:

```
r"^[0-9]{4} [A-Za-z]*"
```

Now we want to improve our regular expression. Let's assume that there is no city name in Switzerland, which consists of less than 3 letters, at least 3 letters. We can denote this by [A-Za-z]{3,}. Now we have to recognize lines with German post code (5 digits) lines as well, i.e. the post code can now consist of either four or five digits:

```
r"^[0-9]{4,5} [A-Z][a-z]{2,}"
```

The general syntax is {from, to}, meaning the expression has to appear at least "from" times and not more than "to" times. {, to} is an abbreviated spelling for {0,to} and {from,} is an abbreviation for "at least from times but no upper limit"

We can group a part of a regular expression by surrounding it with parenthesis (round brackets). This way we can apply operators to the complete group instead of a single character.

Parenthesis (round brackets, braces) are not only group subexpressions but they also create back references. The part of the string matched by the grouped part of the regular expression, i.e. the subexpression in parenthesis, is stored in a back reference. With the aid of back references we can reuse parts of regular expressions. These stored values can be both reused inside the expression itself and afterwards, when the regexpr is executed. Before we continue with our treatise about back references, we want to strew in a paragraph about match objects, which is important for our next examples with back references.

So far we have just checked, if an expression matched or not. We used the fact the re.search() returns a match object if it matches and None otherwise. We haven't been interested e.g. in what has been matched. The match object contains a lot of data about what has been matched, positions and so on.

A match object contains the methods group(), span(), start() and end(), as it can be seen in the following application:

```python
import re
mo = re.search("[0-9]+", "Customer number: 232454, Date: February
12, 2011")
mo.group()
```
Output: `'232454'`

```python
mo.span()
```
Output: `(17, 23)`

```python
mo.start()
```
Output: `17`

```python
mo.end()
```
Output: `23`

```python
mo.span()[0]
```
Output: `17`

```python
mo.span()[1]
```
Output: `23`

These methods are not difficult to understand. span() returns a tuple with the start and end position, i.e. the string index where the regular expression started matching in the string and ended matching. The methods start() and end() are in a way superfluous as the information is contained in span(), i.e. span()[0] is equal to start() and span()[1] is equal to end(). group(), if called without argument, it returns the substring, which had been matched by the complete regular expression. With the help of group() we are also capable of accessing the matched substring by grouping parentheses, to get the matched substring of the n-th group, we call group() with the argument n: group(n). We can also call group with more than integer argument, e.g. group(n,m). group(n,m) - provided there exists a subgoup n and m - returns a tuple with the matched substrings. group(n,m) is equal to (group(n), group(m)):

```python
import re
mo = re.search("([0-9]+).*: (.*)", "Customer number: 232454, Dat
e: February 12, 2011")
mo.group()
```

Output: '232454, Date: February 12, 2011'

```python
mo.group(1)
```

Output: '232454'

```python
mo.group(2)
```

Output: 'February 12, 2011'

```python
mo.group(1,2)
```

Output: ('232454', 'February 12, 2011')

A very intuitive example are XML or HTML tags. E.g. let's assume we have a file (called "tags.txt") with content like this:

```
<composer> Wolfgang Amadeus Mozart </composer>
<author> Samuel Beckett </author>
<city> London </city>
```

We want to rewrite this text automatically to

```
composer: Wolfgang Amadeus Mozart
author: Samuel Beckett
city: London
```

The following little Python script does the trick. The core of this script is the regular expression. This regular expression works like this: It tries to match a less than symbol "<". After this it is reading lower case letters

until it reaches the greater than symbol. Everything encountered within "<" and ">" has been stored in a back reference which can be accessed within the expression by writing \1. Let's assume \1 contains the value "composer". When the expression has reached the first ">", it continues matching, as the original expression had been "(.*)":

```python
import re
fh = open("tags.txt")
for i in fh:
    res = re.search(r"<([a-z]+)>(.*)</\1>",i)
    print(res.group(1) + ": " + res.group(2))
```

```
composer:  Wolfgang Amadeus Mozart
author:  Samuel Beckett
city:  London
```

If there are more than one pair of parenthesis (round brackets) inside the expression, the backreferences are numbered \1, \2, \3, in the order of the pairs of parenthesis.

Exercise: The next Python example makes use of three back references. We have an imaginary phone list of the Simpsons in a list. Not all entries contain a phone number, but if a phone number exists it is the first part of an entry. Then, separated by a blank, a surname follows, which is followed by first names. Surname and first name are separated by a comma. The task is to rewrite this example in the following way:

```
Allison Neu 555-8396
C. Montgomery Burns
Lionel Putz 555-5299
Homer Jay Simpson 555-73347
```

Python script solving the rearrangement problem:

```python
import re

l = ["555-8396 Neu, Allison",
     "Burns, C. Montgomery",
     "555-5299 Putz, Lionel",
     "555-7334 Simpson, Homer Jay"]

for i in l:
    res = re.search(r"([0-9-]*)\s*([A-Za-z]+),\s+(.*)", i)
    print(res.group(3) + " " + res.group(2) + " " + res.group(1))
```

```
Allison Neu 555-8396
C. Montgomery Burns
Lionel Putz 555-5299
Homer Jay Simpson 555-7334
```

In the previous paragraph we introduced "Capturing Groups" and "Back references". More precisely, we could have called them "Numbered Capturing Groups" and "Numbered Backreferences". Using capturing groups instead of "numbered" capturing groups allows you to assign descriptive names instead of automatic numbers to the groups. In the following example, we demonstrate this approach by catching the hours, minutes and seconds from a UNIX date string.

```python
import re
s = "Sun Oct 14 13:47:03 CEST 2012"
expr = r"\b(?P<hours>\d\d):(?P<minutes>\d\d):(?P<seconds>\d\d)\b"
x = re.search(expr,s)
x.group('hours')
```

Output: `'13'`

```python
x.group('minutes')
```

Output: `'47'`

```python
x.start('minutes')
```

Output: `14`

```python
x.end('minutes')
```

Output: `16`

```python
x.span('seconds')
```

Output: `(17, 19)`

COMPREHENSIVE PYTHON EXERCISE

In this comprehensive exercise, we have to bring the information of two files together. In the first file, we have a list of nearly 15000 lines of post codes with the corresponding city names plus additional information. Here are some arbitrary lines of this file:

```
osm_id ort plz bundesland
1104550 Aach 78267 Baden-Württemberg
...
446465 Freiburg (Elbe) 21729 Niedersachsen
62768 Freiburg im Breisgau 79098 Baden-Württemberg
62768 Freiburg im Breisgau 79100 Baden-Württemberg
62768 Freiburg im Breisgau 79102 Baden-Württemberg
...
454863 Fulda 36037 Hessen
```

```
454863 Fulda 36039 Hessen
454863 Fulda 36041 Hessen
...
1451600 Gallin 19258 Mecklenburg-Vorpommern
449887 Gallin-Kuppentin 19386 Mecklenburg-Vorpommern
...
57082 Gärtringen 71116 Baden-Württemberg
1334113 Gartz (Oder) 16307 Brandenburg
...
2791802 Giengen an der Brenz 89522 Baden-Württemberg
2791802 Giengen an der Brenz 89537 Baden-Württemberg
...
1187159 Saarbrücken 66133 Saarland
1256034 Saarburg 54439 Rheinland-Pfalz
1184570 Saarlouis 66740 Saarland
1184566 Saarwellingen 66793 Saarland
```

The other file contains a list of the 19 largest German cities. Each line consists of the rank, the name of the city, the population, and the state (Bundesland):

```
1.  Berlin           3.382.169 Berlin
2.  Hamburg          1.715.392 Hamburg
3.  München          1.210.223 Bayern
4.  Köln               962.884 Nordrhein-Westfalen
5.  Frankfurt am Main 646.550 Hessen
6.  Essen              595.243 Nordrhein-Westfalen
7.  Dortmund           588.994 Nordrhein-Westfalen
8.  Stuttgart          583.874 Baden-Württemberg
9.  Düsseldorf         569.364 Nordrhein-Westfalen
10. Bremen             539.403 Bremen
11. Hannover           515.001 Niedersachsen
12. Duisburg           514.915 Nordrhein-Westfalen
13. Leipzig            493.208 Sachsen
14. Nürnberg           488.400 Bayern
15. Dresden            477.807 Sachsen
16. Bochum             391.147 Nordrhein-Westfalen
17. Wuppertal          366.434 Nordrhein-Westfalen
18. Bielefeld          321.758 Nordrhein-Westfalen
19. Mannheim           306.729 Baden-Württemberg
```

Our task is to create a list with the top 19 cities, with the city names accompanied by the postal code. If you want to test the following program, you have to save the list above in a file called largest_cities_germany.txt and you have to download and save the list of German post codes.

```python
import re
```

```python
with open("zuordnung_plz_ort.txt", encoding="utf-8") as fh_post_co
des:
    codes4city = {}
    for line in fh_post_codes:
        res = re.search(r"[\d ]+([^\d]+[a-z])\s(\d+)", line)
        if res:
            city, post_code = res.groups()
            if city in codes4city:
                codes4city[city].add(post_code)
            else:
                codes4city[city] = {post_code}

with open("largest_cities_germany.txt", encoding="utf-8") as fh_la
rgest_cities:
    for line in fh_largest_cities:
        re_obj = re.search(r"^[0-9]{1,2}\.\s+([\w\s-
]+\w)\s+[0-9]", line)
        city = re_obj.group(1)
        print(city, codes4city[city])
```

Berlin {'12685', '12279', '12353', '12681', '13465', '12163', '103
65', '12351', '13156', '12629', '12689', '12053', '13086', '1205
7', '12157', '10823', '12621', '14129', '13359', '13435', '1096
3', '12209', '12683', '10369', '10787', '12277', '14059', '1245
9', '13593', '12687', '10785', '12526', '10717', '12487', '1220
3', '13583', '12347', '12627', '10783', '12559', '14050', '1096
9', '10367', '12207', '13409', '10555', '10623', '10249', '1031
5', '12109', '12055', '13353', '13509', '10407', '12051', '1215
9', '10629', '13503', '13591', '10961', '10777', '13439', '1305
7', '10999', '12309', '12437', '10781', '12305', '10965', '1216
7', '10318', '10409', '12359', '10779', '14169', '10437', '1216
1', '10589', '13581', '12679', '13505', '12555', '10405', '1011
9', '13355', '10179', '12357', '13469', '13351', '14055', '1408
9', '10627', '10829', '12107', '12435', '10319', '13507', '1055
1', '13437', '13347', '13627', '12524', '13599', '13189', '1312
9', '13467', '12589', '12489', '10825', '13585', '10625', '1252
7', '13405', '10439', '13597', '10317', '10553', '12101', '1055
7', '12355', '13587', '13053', '10967', '12439', '13589', '1070
7', '13629', '13088', '10719', '10243', '12045', '12105', '1234
9', '13125', '10117', '14199', '13187', '13357', '13349', '1340
3', '10178', '12047', '13055', '12165', '12249', '13407', '1216
9', '12587', '14053', '14052', '12103', '10245', '14193', '1220
5', '10827', '10585', '10587', '12557', '10435', '12623', '1071
3', '12059', '13059', '14109', '13051', '13127', '12307', '1071
5', '14057', '14195', '12619', '13158', '12043', '13089', '1071
1', '10997', '12099', '10247', '10709', '13595', '14165', '1224
7', '13159', '14167', '10789', '12049', '14197', '10559', '1011
5', '14163'}
Hamburg {'22589', '22305', '22587', '22395', '20457', '22391', '22
763', '20359', '22393', '22527', '20249', '22605', '21037', '2749
9', '22547', '20097', '22147', '20255', '22765', '22115', '2230
9', '20259', '22767', '20095', '22399', '22113', '22087', '2112
9', '22299', '22159', '22083', '22455', '20144', '21033', '2211
9', '20354', '22145', '22041', '22049', '21035', '21077', '2009
9', '20459', '20149', '21031', '22453', '20539', '21039', '2229
7', '22149', '20251', '22607', '22529', '22179', '21075', '2233
9', '22397', '22335', '22047', '22175', '22085', '21079', '2230
3', '22111', '22525', '21107', '20537', '22143', '21029', '2208
1', '20253', '22301', '22337', '21109', '22459', '20355', '2255
9', '22549', '22761', '21149', '20357', '22415', '22307', '2217
7', '22045', '20146', '22457', '22089', '22417', '22769', '2241
9', '22117', '22609', '20535', '21073', '20257', '22523', '2235
9', '20148', '22043', '21147'}
München {'81241', '81669', '80336', '80796', '80799', '80339', '80
539', '80689', '80939', '81929', '80634', '81249', '81545', '8046
9', '81475', '80803', '81371', '81373', '81476', '80686', '8099

9', '81667', '81927', '80804', '81739', '80997', '80687', '8079
8', '80933', '81477', '81541', '81369', '81379', '81549', '8167
5', '80637', '80801', '80992', '80993', '81377', '81543', '8167
7', '81925', '81479', '81247', '80337', '81673', '85540', '8080
2', '80797', '80333', '80638', '81827', '81671', '80809', '8063
9', '81547', '81825', '81245', '81829', '81243', '81375', '8173
7', '80805', '80335', '81679', '80331', '80636', '80935', '8053
8', '80807', '80937', '81539', '80995', '81735'}
Köln {'50939', '50827', '50668', '50676', '50999', '51065', '5096
8', '50733', '50937', '50769', '51147', '50670', '51061', '5082
3', '50767', '50829', '50674', '51069', '50735', '50672', '5067
9', '50737', '50931', '50678', '50969', '51103', '50667', '5085
8', '50859', '50933', '51105', '51467', '51063', '51107', '5082
5', '50996', '51109', '51145', '50935', '50765', '51067', '5099
7', '51149', '50739', '50677', '51143'}
Frankfurt am Main {'60437', '60311', '60439', '60329', '60549', '6
0320', '60322', '60486', '65934', '60489', '60599', '60438', '6043
1', '60594', '60529', '60488', '60386', '60310', '60306', '6032
5', '60313', '60598', '65931', '60389', '60308', '65936', '6032
7', '60435', '60385', '60316', '60596', '60433', '60318', '6052
8', '65933', '60487', '60314', '60388', '60323', '65929', '60326'}
Essen {'45259', '45276', '45327', '45145', '45138', '45355', '4521
9', '45257', '45326', '45141', '45128', '45277', '45130', '4528
9', '45279', '45139', '45307', '45356', '45136', '45144', '4523
9', '45357', '45147', '45131', '45359', '45127', '45329', '4530
9', '45134', '45143', '45133', '45149'}
Dortmund {'44139', '44263', '44141', '44267', '44265', '44319', '4
4145', '44289', '44339', '44229', '44309', '44357', '44147', '4437
9', '44143', '44149', '44227', '44328', '44135', '44359', '4413
7', '44329', '44269', '44287', '44369', '44388', '44225'}
Stuttgart {'70191', '70186', '70193', '70195', '70176', '70378',
'70197', '70327', '70435', '70192', '70376', '70182', '70174', '70
569', '70190', '70188', '70597', '70374', '70567', '70499', '7059
9', '70184', '70329', '70178', '70565', '70563', '70439', '7062
9', '70469', '70619', '70199', '70372', '70173', '70180', '70437'}
Düsseldorf {'40227', '40215', '40549', '40225', '40593', '40476',
'40625', '40231', '40595', '40221', '40217', '40229', '40470', '40
721', '40489', '40627', '40479', '40212', '40211', '40219', '4023
5', '40547', '40223', '40477', '40629', '40233', '40599', '4058
9', '40597', '40213', '40237', '40472', '40474', '40468', '4059
1', '40210', '40545', '40239'}
Bremen {'28195', '28201', '28215', '27568', '28759', '28307', '287
55', '28217', '28279', '28777', '28213', '28719', '28325', '2819
7', '28779', '28757', '28209', '28309', '28207', '28219', '2827
7', '28203', '28199', '28327', '28237', '28205', '28211', '2832
9', '28717', '28357', '28359', '28355', '28239', '28259'}

```
Hannover {'30459', '30519', '30627', '30669', '30539', '30419', '3
0177', '30521', '30629', '30559', '30453', '30655', '30659', '3016
9', '30455', '30449', '30179', '30175', '30625', '30159', '3045
1', '30171', '30457', '30657', '30165', '30173', '30161', '3016
3', '30167'}
Duisburg {'47249', '47229', '47055', '47179', '47138', '47139', '4
7198', '47057', '47166', '47199', '47279', '47169', '47259', '4726
9', '47228', '47137', '47226', '47119', '47051', '47167', '4705
8', '47059', '47239', '47178', '47053'}
Leipzig {'4155', '4289', '4357', '4205', '4349', '4229', '4288',
'4275', '4209', '4179', '4347', '4328', '4177', '4129', '4319', '4
178', '4318', '4277', '4315', '4103', '4105', '4317', '4249', '431
6', '4158', '4329', '4109', '4356', '4279', '4107', '4207', '415
9', '4157', '4299'}
Nürnberg {'90480', '90482', '90427', '90475', '90411', '90471', '9
0439', '90403', '90478', '90451', '90443', '90431', '90453', '9047
3', '90459', '90469', '90402', '90408', '90455', '90489', '9044
1', '90449', '90461', '90419', '90491', '90425', '90409', '90429'}
Dresden {'1159', '1139', '1326', '1187', '1156', '1069', '1108',
'1099', '1465', '1109', '1127', '1169', '1324', '1328', '1259', '1
277', '1239', '1257', '1237', '1217', '1157', '1097', '1219', '130
7', '1309', '1129', '1189', '1279', '1067'}
Bochum {'44869', '44809', '44799', '44787', '44803', '44867', '448
01', '44866', '44795', '44894', '44793', '44805', '44807', '4479
1', '44879', '44797', '44892', '44789'}
Wuppertal {'42389', '42399', '42327', '42115', '42107', '42279',
'42109', '42103', '42369', '42111', '42275', '42117', '42113', '42
283', '42281', '42329', '42105', '42289', '42119', '42277', '4228
7', '42349', '42285'}
Bielefeld {'33613', '33699', '33729', '33611', '33602', '33615',
'33739', '33605', '33659', '33609', '33689', '33617', '33647', '33
619', '33719', '33607', '33604', '33649'}
Mannheim {'68199', '68309', '68239', '68305', '68161', '68167', '6
8165', '68229', '68159', '68307', '68163', '68259', '68169', '6821
9'}
```

We want to present another real life example in our Python course. A regular expression for UK postcodes. We write an expression, which is capable of recognizing the postal codes or postcodes of the UK.

Postcode units consist of between five and seven characters, which are separated into two parts by a space. The two to four characters before the space represent the so-called outward code or out code intended to directly mail from the sorting office to the delivery office. The part following the space, which consists of a digit followed by two uppercase characters, comprises the so-called inward code, which is needed to sort mail

at the final delivery office. The last two uppercase characters do not use the letters CIKMOV, so as not to resemble digits or each other when hand-written.

The outward code can have the following form: One or two uppercase characters, followed by either a digit or the letter R, optionally followed by an uppercase character or a digit. (We do not consider all the detailed rules for postcodes, i.e only certain character sets are valid depending on the position and the context.)A regular expression for matching this superset of UK postcodes looks like this:

```
r"\b[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}\b"
```

The following Python program uses the regexp above:

```python
import re

example_codes = ["SW1A 0AA", # House of Commons
                 "SW1A 1AA", # Buckingham Palace
                 "SW1A 2AA", # Downing Street
                 "BX3 2BB", # Barclays Bank
                 "DH98 1BT", # British Telecom
                 "N1 9GU", # Guardian Newspaper
                 "E98 1TT", # The Times
                 "TIM E22", # a fake postcode
                 "A B1 A22", # not a valid postcode
                 "EC2N 2DB", # Deutsche Bank
                 "SE9 2UG", # University of Greenwhich
                 "N1 0UY", # Islington, London
                 "EC1V 8DS", # Clerkenwell, London
                 "WC1X 9DT", # WC1X 9DT
                 "B42 1LG", # Birmingham
                 "B28 9AD", # Birmingham
                 "W12 7RJ", # London, BBC News Centre
                 "BBC 007" # a fake postcode
                 ]

pc_re = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"

for postcode in example_codes:
    r = re.search(pc_re, postcode)
    if r:
        print(postcode + " matched!")
    else:
        print(postcode + " is not a valid postcode!")
```

```
SW1A 0AA matched!
SW1A 1AA matched!
SW1A 2AA matched!
BX3 2BB matched!
DH98 1BT matched!
N1 9GU matched!
E98 1TT matched!
TIM E22 is not a valid postcode!
A B1 A22 is not a valid postcode!
EC2N 2DB matched!
SE9 2UG matched!
N1 0UY matched!
EC1V 8DS matched!
WC1X 9DT matched!
B42 1LG matched!
B28 9AD matched!
W12 7RJ matched!
BBC 007 is not a valid postcode!
```

ADVANCED REGULAR EXPRESSIONS

INTRODUCTION

In our introduction to regular expressions of our tutorial we have covered the basic principles of regular expressions. We have shown, how the simplest regular expression looks like. We have also learnt, how to use regular expressions in Python by using the search() and the match() methods of the re module. The concept of formulating and using character classes should be well known by now, as well as the predefined character classes like \d, \D, \s, \S, and so on. You must have learnt how to match the beginning and the end of a string with a regular expression. You must know the special meaning of the question mark to make items optional. We have also introduced the quantifiers to repeat characters and groups arbitrarily or in certain ranges.

You must also be familiar with the use of grouping and the syntax and usage of back references.

Furthermore, we had explained the match objects of the re module and the information they contain and how to retrieve this information by using the methods span(), start(), end(), and group().

The introduction ended with a comprehensive example in Python.

In this chapter we will continue with our explanations about the syntax of the regular expressions. We will also explain further methods of the Python module re. E.g. how to find all the matched substrings of a regular expression. A task which needs programming in other programming languages like Perl or Java, but can be dealt with the call of one method of the re module of Python. So far, we only know how to define a choice of characters with a character class. We will demonstrate how to formulate alternations of substrings in this chapter of our tutorial,

FINDING ALL MATCHED SUBSTRINGS

The Python module re provides another great method, which other languages like Perl and Java don't provide. If you want to find all the substrings in a string, which match a regular expression, you have to use a loop in Perl and other languages, as can be seen in the following Perl snippet:

```
while ($string =~ m/regex/g) {
  print "Found '$&'.  Next attempt at character " . pos($string)+1
. "\n";
}
```

It's a lot easier in Python. No need to loop. We can just use the findall method of the re module:

```
     re.findall(pattern, string[, flags])
```

findall returns all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order in which they are found.

```
t="A fat cat doesn't eat oat but a rat eats bats."
mo = re.findall("[force]at", t)
print(mo)
```

```
['fat', 'cat', 'eat', 'oat', 'rat', 'eat']
```

If one or more groups are present in the pattern, findall returns a list of groups. This will be a list of tuples if the pattern has more than one group. We demonstrate this in our next example. We have a long string with various Python training courses and their dates. With the first call to findall, we don't use any grouping and receive the complete string as a result. In the next call, we use grouping and findall returns a list of 2-tuples, each having the course name as the first component and the dates as the second component:

```
import re
courses = "Python Training Course for Beginners: 15/Aug/2011 - 19/
Aug/2011;Python Training Course Intermediate: 12/Dec/2011 - 16/De
c/2011;Python Text Processing Course:31/Oct/2011 - 4/Nov/2011"
items = re.findall("[^:]*:[^;]*;?", courses)
items
```

Output:
```
['Python Training Course for Beginners: 15/Aug/2011 - 19/Aug/
2011;',
 'Python Training Course Intermediate: 12/Dec/2011 - 16/Dec/2
011;',
 'Python Text Processing Course:31/Oct/2011 - 4/Nov/2011']
```

```
items = re.findall("([^:]*):([^;]*;?)", courses)
items
```

Output:
```
[('Python Training Course for Beginners', ' 15/Aug/2011 - 19/
Aug/2011;'),
 ('Python Training Course Intermediate', ' 12/Dec/2011 - 16/D
ec/2011;'),
 ('Python Text Processing Course', '31/Oct/2011 - 4/Nov/201
1')]
```

ALTERNATIONS

In our introduction to regular expressions we had introduced character classes. Character classes offer a choice out of a set of characters. Sometimes we need a choice between several regular expressions. It's a logical "or" and that's why the symbol for this construct is the "|" symbol. In the following example, we check, if one of

the cities London, Paris, Zurich, Konstanz Bern or Strasbourg appear in a string preceded by the word "location":

```python
import re
str = "Course location is London or Paris!"
mo = re.search(r"location.*(London|Paris|Zurich|Strasbourg)",str)
if mo: print(mo.group())
```

```
location is London or Paris
```

If you consider the previous example as too artificial, here is another one. Let's assume, you want to filter your email. You want to find all the correspondence (conversations) between you and Guido van Rossum, the creator and designer of Python. The following regular expression is helpful for this purpose:

```
r"(^To:|^From:) (Guido|van Rossum)"
```

This expression matches all lines starting with either 'To:' or 'From:', followed by a space and then either by the first name 'Guido' or the surname 'van Rossum'.

COMPILING REGULAR EXPRESSIONS

If you want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.

The general syntax:

```
re.compile(pattern[, flags])
```

compile returns a regex object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

Abbreviation	Full name	Description
re.I	re.IGNORECASE	Makes the regular expression case-insensitive
re.L	re.LOCALE	The behaviour of some special sequences like \w, \W, \b,\s, \S will be made dependent on the current locale, i.e. the user's language, country etc.
re.M	re.MULTILINE	^ and $ will match at the beginning and at the end of each line and not just at the beginning and the end of the string
re.S	re.DOTALL	The dot "." will match every character **plus the newline**
re.U	re.UNICODE	Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties

re.X	re.VERBOSE	Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class. # are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character can be used to start a comment.

Compiled regular objects usually are not saving much time, because Python internally compiles AND CACHES regexes whenever you use them with re.search() or re.match(). The only extra time a non-compiled regex takes is the time it needs to check the cache, which is a key lookup of a dictionary.

A good reason to use them is to separate the definition of a regex from its use.

<div style="background-color:orange; height:40px;"></div>

We have already introduced a regular expression for matching a superset of UK postcodes in our introductory chapter:

    r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"

We demonstrate with this regular expression, how we can use the compile functionality of the module re in the following interactive session. The regular expression "regex" is compiled with re.compile(regex) and the compiled object is saved in the object compiled_re. Now we call the method search() of the object compiled_re:

```python
import re
regex = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
address = "BBC News Centre, London, W12 7RJ"
compiled_re = re.compile(regex)
res = compiled_re.search(address)
print(res)
```

```
<re.Match object; span=(25, 32), match='W12 7RJ'>
```
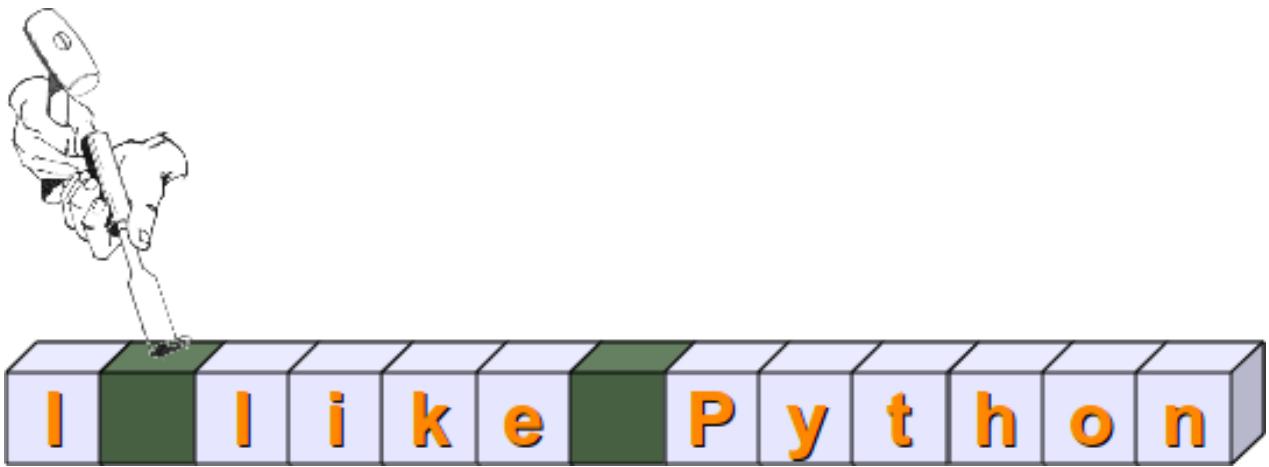
SPLITTING A STRING WITH OR WITHOUT REGULAR EXPRESSIONS

There is a string method split, which can be used to split a string into a list of substrings.

    str.split([sep[, maxsplit]])

As you can see, the method split has two optional parameters. If none is given (or is None) , a string will be separated into substring using whitespaces as delimiters, i.e. every substring consisting purely of whitespaces is used as a delimiter.

We demonstrate this behaviour with a famous quotation by Abraham Lincoln:

```
law_courses = "Let reverence for the laws be breathed by every Ame
rican mother to the lisping babe that prattles on her lap. Let it
be taught in schools, in seminaries, and in colleges. Let it be wr
itten in primers, spelling books, and in almanacs. Let it be preac
hed from the pulpit, proclaimed in legislative halls, and enforce
d in the courts of justice. And, in short, let it become the polit
ical religion of the nation."
law_courses.split()
```

```
['Let',
 'reverence',
 'for',
 'the',
 'laws',
 'be',
 'breathed',
 'by',
 'every',
 'American',
 'mother',
 'to',
 'the',
 'lisping',
 'babe',
 'that',
 'prattles',
 'on',
 'her',
 'lap.',
 'Let',
 'it',
 'be',
 'taught',
 'in',
 'schools,',
 'in',
 'seminaries,',
 'and',
 'in',
 'colleges.',
 'Let',
 'it',
 'be',
 'written',
 'in',
 'primers,',
 'spelling',
 'books,',
 'and',
 'in',
 'almanacs.',
 'Let',
 'it',
 'be',
 'preached',
```

```
      'from',
      'the',
      'pulpit,',
      'proclaimed',
      'in',
      'legislative',
      'halls,',
      'and',
      'enforced',
      'in',
      'the',
      'courts',
      'of',
      'justice.',
      'And,',
      'in',
      'short,',
      'let',
      'it',
      'become',
      'the',
      'political',
      'religion',
      'of',
      'the',
      'nation.']
```

Now we look at a string, which could stem from an Excel or an OpenOffice calc file. We have seen in our previous example that split takes whitespaces as default separators. We want to split the string in the following little example using semicolons as separators. The only thing we have to do is to use ";" as an argument of split():

```
line = "James;Miller;teacher;Perl"
line.split(";")
```

Output: `['James', 'Miller', 'teacher', 'Perl']`

The method split() has another optional parameter: maxsplit. If maxsplit is given, at most maxsplit splits are done. This means that the resulting list will have at most "maxsplit + 1" elements. We will illustrate the mode of operation of maxsplit in the next example:

```
mammon = "The god of the world's leading religion. The chief templ
e is in the holy city of New York."
mammon.split(" ",3)
```

```
['The',
 'god',
 'of',
 "the world's leading religion. The chief temple is in the ho
ly city of New York."]
```

We used a Blank as a delimiter string in the previous example, which can be a problem: If multiple blanks or whitespaces are connected, split() will split the string after every single blank, so that we will get empty strings and strings with only a tab inside ('\t') in our result list:

```
mammon = "The god  \t of the world's leading religion. The chief t
emple is in the holy city of New York."
mammon.split(" ",5)
```

Output:
```
['The',
 'god',
 '',
 '\t',
 'of',
 "the world's leading religion. The chief temple is in the ho
ly city of New York."]
```

We can prevent the separation of empty strings by using None as the first argument. Now split will use the default behaviour, i.e. every substring consisting of connected whitespace characters will be taken as one separator:

```
mammon.split(None,5)
```

Output:
```
['The',
 'god',
 'of',
 'the',
 "world's",
 'leading religion. The chief temple is in the holy city of N
ew York.']
```

The string method split() is the right tool in many cases, but what, if you want e.g. to get the bare words of a text, i.e. without any special characters and whitespaces. If we want this, we have to use the split function from the re module. We illustrate this method with a short text from the beginning of Metamorphoses by Ovid:

```
import re
metamorphoses = "OF bodies chang'd to various forms, I sing: Ye Go
```

```
ds, from whom these miracles did spring, Inspire my numbers with c
oelestial heat;"
re.split("\W+",metamorphoses)
```

Output:
```
['OF',
 'bodies',
 'chang',
 'd',
 'to',
 'various',
 'forms',
 'I',
 'sing',
 'Ye',
 'Gods',
 'from',
 'whom',
 'these',
 'miracles',
 'did',
 'spring',
 'Inspire',
 'my',
 'numbers',
 'with',
 'coelestial',
 'heat',
 '']
```

The following example is a good case, where the regular expression is really superior to the string split. Let's assume that we have data lines with surnames, first names and professions of names. We want to clear the data line of the superfluous and redundant text descriptions, i.e. "surname: ", "prename: " and so on, so that we have solely the surname in the first column, the first name in the second column and the profession in the third column:

```
import re
lines = ["surname: Obama, prename: Barack, profession: presiden
t", "surname: Merkel, prename: Angela, profession: chancellor"]
for line in lines:
    print(re.split(",* *\w*: ", line))

['', 'Obama', 'Barack', 'president']
['', 'Merkel', 'Angela', 'chancellor']
```

We can easily improve the script by using a slice operator, so that we don't have the empty string as the first

element of our result lists:

```python
import re
lines = ["surname: Obama, prename: Barack, profession: presiden
t", "surname: Merkel, prename: Angela, profession: chancellor"]
for line in lines:
    print(re.split(",* *\w*: ", line)[1:])
```

```
['Obama', 'Barack', 'president']
['Merkel', 'Angela', 'chancellor']
```

And now for something completely different: There is a connection between Barack Obama and Python, or better Monty Python. John Cleese, one of the members of Monty Python told the Western Daily Press in April 2008: "I'm going to offer my services to him as a speech writer because I think he is a brilliant man."

SEARCH AND REPLACE WITH SUB

re.sub(regex, replacement, subject)

Every match of the regular expression regex in the string subject will be replaced by the string replacement. Example:

```python
import re
str = "yes I said yes I will Yes."
res = re.sub("[yY]es","no", str)
print(res)
```

```
no I said no I will no.
```

EXERCISES

EXERCISE 1

Given is a file with the name 'order_journal.txt' in the following format:

```
%%writefile order_journal.txt
customer-id 1289
T83456
customer-id 1289
customer-id 1205
T10032
B77301
customer-id 1205
```

```
customer-id 1410
K34001
T98987
customer-id 1410
customer-id 1205
T10786
C77502
customer-id 1205
customer-id 1289
Z22334
customer-id 1289
```

```
Overwriting order_journal.txt
```

Write a new file 'order_journal_regrouped.txt', in which the data is regrouped in the following way:

```
1289,T83456
1289,Z22334
1205,T10032
1205,B77301
1205,T10786
1205,C77502
1410,K34001
1410,T98987
```

SOLUTIONS

SOLUTION TO EXERCISE 1

```python
import re

txt = open("order_journal.txt").read()

data = {}     # will contain the chustomer-id as keys and the oder
s as a value string
for x in re.finditer(r"customer-id ([\d\n]{4})(.*?)customer-id
\1", txt, re.DOTALL):
    key, values = x.groups()

    if key in data:
        data[key] += values
    else:
        data[key] = values
with open("order_journal_regrouped.txt", "w") as fh:
    for key in data:
```

```
        for art_no in data[key].split():
            fh.write(f"{key},{art_no}\n")
```
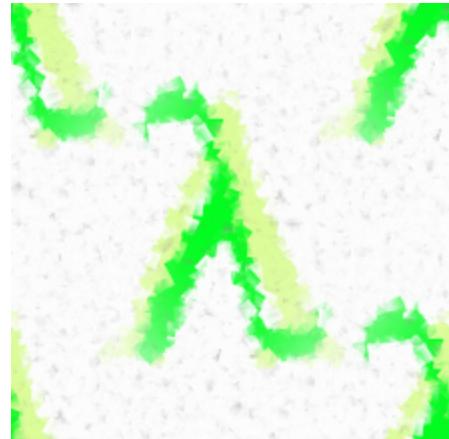
We can check the content of the newly created file:

```
content = open("order_journal_regrouped.txt").read()
print(content)
```

```
1289,T83456
1289,Z22334
1205,T10032
1205,B77301
1205,T10786
1205,C77502
1410,K34001
1410,T98987
```

LAMBDA, FILTER, REDUCE AND MAP

If Guido van Rossum, the author of the programming language Python, had got his will, this chapter would have been missing in our tutorial. In his article from May 2005 "All Things Pythonic: The fate of reduce() in Python 3000", he gives his reasons for dropping lambda, map(), filter() and reduce(). He expected resistance from the Lisp and the scheme "folks". What he didn't anticipate was the rigidity of this opposition. Enough that Guido van Rossum wrote hardly a year later: "After so many attempts to come up with an alternative for lambda, perhaps we should admit defeat. I've not had the time to follow the most recent rounds, but I propose that we keep lambda, so as to stop wasting everybody's talent and time on an impossible quest." We can see the result: lambda, map() and filter() are still part of core Python. Only reduce() had to go; it moved into the module functools.

His reasoning for dropping them is like this:

- There is an equally powerful alternative to lambda, filter, map and reduce, i.e. list comprehension
- List comprehension(is more evident and easier to understand
- Having both list comprehension and "Filter, map, reduce and lambda" is transgressing the Python motto "There should be one obvious way to solve a problem"

Some like it, others hate it and many are afraid of the lambda operator. The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers.

The general syntax of a lambda function is quite simple:

lambda argument_list: expression

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.

The following example of a lambda function returns the sum of its two arguments:

```
sum = lambda x, y : x + y
sum(3,4)
```

Output: 7

The above example might look like a plaything for a mathematician. A formalism which turns an easy to comprehend issue into an abstract harder to grasp formalism. Above all, we could have had the same effect by just using the following conventional function definition:

```python
def sum(x,y):
    return x + y

sum(3,4)
```

Output: 7

We can assure you that the advantages of this approach will be apparent, when you will have learnt to use the map() function.

THE MAP() FUNCTION

As we have mentioned earlier, the advantage of the lambda operator can be seen when it is used in combination with the map() function. map() is a function which takes two arguments:

r = map(func, seq)

The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq. Before Python3, map() used to return a list, where each element of the result list was the result of the function func applied on the corresponding element of the list or tuple "seq". With Python 3, map() returns an iterator.

The following example illustrates the way of working of map():

```python
def fahrenheit(T):
    return ((float(9)/5)*T + 32)

def celsius(T):
    return (float(5)/9)*(T-32)

temperatures = (36.5, 37, 37.5, 38, 39)
F = map(fahrenheit, temperatures)
C = map(celsius, F)
temperatures_in_Fahrenheit = list(map(fahrenheit, temperatures))
temperatures_in_Celsius = list(map(celsius, temperatures_in_Fahrenheit))
print(temperatures_in_Fahrenheit)
print(temperatures_in_Celsius)

[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
[36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]
```

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

```
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
print(F)
```

```
[102.56, 97.7, 99.14, 100.4, 100.03999999999999]
```

```
C = list(map(lambda x: (float(5)/9)*(x-32), F))
print(C)
```

```
[39.2, 36.5, 37.300000000000004, 38.00000000000001, 37.8]
```

map() can be applied to more than one list. The lists don't have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

```
a = [1, 2, 3, 4]
b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]
```

```
list(map(lambda x, y, z : x+y+z, a, b, c))
```
Output: `[17, 10, 19, 23]`

```
list(map(lambda x, y : x+y, a, b))
```
Output: `[18, 14, 14, 14]`

```
list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
```
Output: `[37.5, 33.0, 24.5, 21.0]`

If one list has fewer elements than the others, map will stop when the shortest list has been consumed:

```
a = [1, 2, 3]
b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]
```

```
list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
```
Output: `[37.5, 33.0, 24.5]`

We can see in the example above that the parameter x gets its values from the list a, while y gets its values

from b, and z from list c.

MAPPING A LIST OF FUNCTIONS

The map function of the previous chapter was used to apply one function to one or more iterables. We will now write a function which applies a bunch of functions, which may be an iterable such as a list or a tuple, for example, to one Python object.

```python
from math import sin, cos, tan, pi

def map_functions(x, functions):
    """ map an iterable of functions on the the object x """
    res = []
    for func in functions:
        res.append(func(x))
    return res

family_of_functions = (sin, cos, tan)
print(map_functions(pi, family_of_functions))
```

```
[1.2246467991473532e-16, -1.0, -1.2246467991473532e-16]
```

The previously defined map_functions function can be simplified with the list comprehension technique, which we will cover in the chapter list comprehension:

```python
def map_functions(x, functions):
    return [ func(x) for func in functions ]
```

FILTERING

The function

filter(function, sequence)

offers an elegant way to filter out all the elements of a sequence "sequence", for which the function function returns True. i.e. an item will be produced by the iterator result of filter(function, sequence) if item is included in the sequence "sequence" and if function(item) returns True.

In other words: The function filter(f,l) needs a function f as its first argument. f has to return a Boolean value, i.e. either True or False. This function will be applied to every element of the list l. Only if f returns True will the element be produced by the iterator, which is the return value of filter(function, sequence).

In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers = list(filter(lambda x: x % 2, fibonacci))
print(odd_numbers)
```

```
[1, 1, 3, 5, 13, 21, 55]
```

```
even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))
print(even_numbers)
```

```
[0, 2, 8, 34]
```

```
even_numbers = list(filter(lambda x: x % 2 -1, fibonacci))
print(even_numbers)
```

```
[0, 2, 8, 34]
```

REDUCING A LIST

As we mentioned in the introduction of this chapter of our tutorial. reduce() had been dropped from the core of Python when migrating to Python 3. Guido van Rossum hates reduce(), as we can learn from his statement in a posting, March 10, 2005, in artima.com:

```
"So now reduce(). This is actually the one I've always hated most,
because, apart from a few examples involving + or *, almost every t
ime I see a reduce() call with a non-trivial function argument, I n
eed to grab pen and paper to diagram what's actually being fed int
o that function before I understand what the reduce() is supposed t
o do. So in my mind, the applicability of reduce() is pretty much l
imited to associative operators, and in all other cases it's bette
r to write out the accumulation loop explicitly."
```

The function

reduce(func, seq)

continually applies the function func() to the sequence seq. It returns a single value.

If seq = [s1, s2, s3, ... , sn], calling reduce(func, seq) works like this:

```
At first the first two elements of seq will be applied to func,
i.e. func(s1,s2) The list on which reduce() works looks now like th
is: [ func(s1, s2), s3, ... , sn ]
In the next step func will be applied on the previous result and th
e third element of the list, i.e. func(func(s1, s2),s3)
The list looks like this now: [ func(func(s1, s2),s3), ... , sn ]
Continues like this until just one element is left and returns thi
s element as the result of reduce()
```

If n is equal to 4 the previous explanation can be illustrated like this:
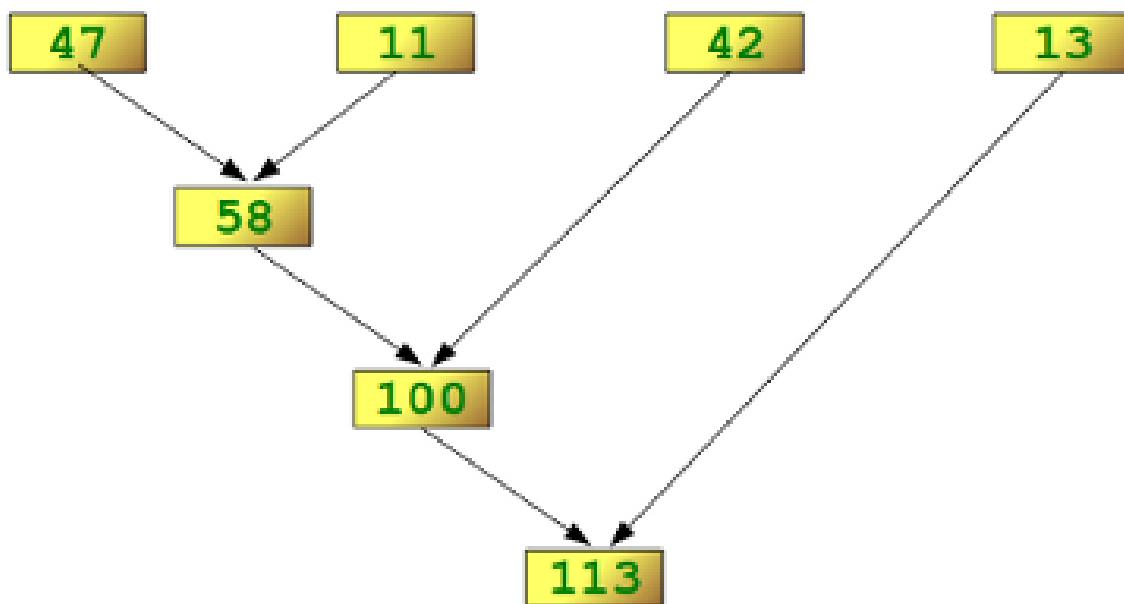
$$[s_1, \quad s_2, \quad s_3, \quad s_4]$$

$$\text{func}(s_1, s_2)$$

$$\text{func}(\text{func}(s_1, s_2), s_3)$$

$$\text{func}(\text{func}(\text{func}(s_1, s_2), s_3), s_4)$$

We want to illustrate this way of working of reduce() with a simple example. We have to import functools to be capable of using reduce:

```python
import functools
functools.reduce(lambda x,y: x+y, [47,11,42,13])
```

Output: 113

The following diagram shows the intermediate steps of the calculation:

EXAMPLES OF REDUCE()

Determining the maximum of a list of numerical values by using reduce:

```
from functools import reduce
f = lambda a,b: a if (a > b) else b
reduce(f, [47,11,42,102,13])
```

Output: 102

Calculating the sum of the numbers from 1 to 100:

```
from functools import reduce
reduce(lambda x, y: x+y, range(1,101))
```

Output: 5050

It's very simple to change the previous example to calculate the product (the factorial) from 1 to a number, but do not choose 100. We just have to turn the "+" operator into "*":

```
reduce(lambda x, y: x*y, range(1,49))
```

Output: 124139155925360726708622890473733750385214863546777600000000000

If you are into lottery, here are the chances to win a 6 out of 49 drawing:

```
reduce(lambda x, y: x*y, range(44,50))/reduce(lambda x, y: x*y, range(1,7))
```

Output: 13983816.0

EXERCISES

1) Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Order Number	Book Title and Author	Quantity	Price per Item
34587	Learning Python, Mark Lutz	4	40.95
98762	Programming Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95
88112	Einführung in Python3, Bernd Klein	3	2

```
    4.99
```

Write a Python program, which returns a list with 2-tuples. Each tuple consists of a the order number and the product of the price per items and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.
Write a Python program using lambda and map.

2) The same bookshop, but this time we work on a different list. The sublists of our lists look like this: [ordernumber, (article number, quantity, price per unit), ... (article number, quantity, price per unit)] Write a program which returns a list of two tuples with (order number, total amount of order).

SOLUTIONS TO THE EXERCISES

```python
orders = [ ["34587", "Learning Python, Mark Lutz", 4, 40.95],
           ["98762", "Programming Python, Mark Lutz", 5, 56.80],
           ["77226", "Head First Python, Paul Barry", 3,32.95],
           ["88112", "Einführung in Python3, Bernd Klei
n",        3, 24.99]]

min_order = 100
invoice_totals = list(map(lambda x: x if x[1] >= min_order else
(x[0], x[1] + 10),
                map(lambda x: (x[0],x[2] * x[3]), orders)))

print(invoice_totals)
```

```
[('34587', 163.8), ('98762', 284.0), ('77226', 108.8500000000000
1), ('88112', 84.97)]
```

```python
from functools import reduce

orders = [ [1, ("5464", 4, 9.99), ("8274",18,12.99), ("9744", 9, 4
4.95)],
           [2, ("5464", 9, 9.99), ("9744", 9, 44.95)],
           [3, ("5464", 9, 9.99), ("88112", 11, 24.99)],
           [4, ("8732", 7, 11.99), ("7733",11,18.99), ("88112",
5, 39.95)] ]

min_order = 100
invoice_totals = list(map(lambda x: [x[0]] + list(map(lambda y:
y[1]*y[2], x[1:])), orders))
invoice_totals = list(map(lambda x: [x[0]] + [reduce(lambda a,b:
```

```
a + b, x[1:])], invoice_totals))
invoice_totals = list(map(lambda x: x if x[1] >= min_order else
(x[0], x[1] + 10), invoice_totals))

print (invoice_totals)
```

```
[[1, 678.3299999999999], [2, 494.46000000000004], [3, 364.79999999
999995], [4, 492.57]]
```

LIST COMPREHENSION

INTRODUCTION

We learned in the previous chapter "Lambda Operator, Filter, Reduce and Map" that Guido van Rossum prefers list comprehensions to constructs using map, filter, reduce and lambda. In this chapter we will cover the essentials about list comprehensions.

List comprehensions were added with Python 2.0. Essentially, it is Python's way of implementing a well-known notation for sets as used by mathematicians. In mathematics the square numbers of the natural numbers are, for example, created by { x^2 | x ∈ ℕ } or the set of complex integers { (x,y) | x ∈ ℤ ∧ y ∈ ℤ }.

List comprehension is an elegant way to define and create lists in Python. These lists have often the qualities of sets, but are not necessarily sets.

List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce(). For most people the syntax of list comprehension is easier to be grasped.

EXAMPLES

In the chapter on lambda and map() we designed a map() function to convert Celsius values into Fahrenheit and vice versa. It looks like this with list comprehension:

```python
Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
print(Fahrenheit)
```

```
[102.56, 97.7, 99.14, 100.03999999999999]
```

A Pythagorean triple consists of three positive integers a, b, and c, such that $a^2 + b^2 = c^2$. Such a triple is commonly written (a, b, c), and the best known example is (3, 4, 5). The following list comprehension creates the Pythagorean triples:

```python
[(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x**2 + y**2 == z**2]
```

```
 [(3, 4, 5),
  (5, 12, 13),
  (6, 8, 10),
  (7, 24, 25),
  (8, 15, 17),
  (9, 12, 15),
  (10, 24, 26),
  (12, 16, 20),
  (15, 20, 25),
  (20, 21, 29)]
```

Another example: Let A and B be two sets, the cross product (or Cartesian product) of A and B, written A×B, is the set of all pairs wherein the first element is a member of the set A and the second element is a member of the set B.

Mathematical definition: A×B = {(a, b) : a belongs to A, b belongs to B}. It's easy to be accomplished in Python:

```python
colours = [ "red", "green", "yellow", "blue" ]
things = [ "house", "car", "tree" ]
coloured_things = [ (x,y) for x in colours for y in things ]
print(coloured_things)
```

```
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'hou
se'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house'),
('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blu
e', 'car'), ('blue', 'tree')]
```

GENERATOR COMPREHENSION

Generator comprehensions were introduced with Python 2.6. They are simply like a list comprehension but with parentheses - round brackets - instead of (square) brackets around it. Otherwise, the syntax and the way of working is like list comprehension, but a generator comprehension returns a generator instead of a list.

```python
x = (x **2 for x in range(20))
print(x)
x = list(x)
print(x)
```

```
<generator object <genexpr> at 0x000002441374D1C8>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 22
5, 256, 289, 324, 361]
```

A MORE DEMANDING EXAMPLE

Calculation of the prime numbers between 1 and 100 using the sieve of Eratosthenes:

```python
noprimes = [j for i in range(2, 8) for j in range(i*2, 100, i)]
primes = [x for x in range(2, 100) if x not in noprimes]
print(primes)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
1, 67, 71, 73, 79, 83, 89, 97]
```

We want to bring the previous example into more general form, so that we can calculate the list of prime numbers up to an arbitrary number n:

```python
from math import sqrt
n = 100
sqrt_n = int(sqrt(n))
no_primes = [j for i in range(2, sqrt_n+1) for j in range(i*2, n,
i)]
no_primes
```

Output:
```
[4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
 38,
 40,
 42,
 44,
 46,
 48,
 50,
 52,
 54,
 56,
 58,
 60,
 62,
 64,
 66,
 68,
 70,
 72,
 74,
 76,
 78,
 80,
 82,
 84,
 86,
 88,
 90,
 92,
 94,
```

```
96,
98,
6,
9,
12,
15,
18,
21,
24,
27,
30,
33,
36,
39,
42,
45,
48,
51,
54,
57,
60,
63,
66,
69,
72,
75,
78,
81,
84,
87,
90,
93,
96,
99,
8,
12,
16,
20,
24,
28,
32,
36,
40,
44,
48,
52,
```

```
56,
60,
64,
68,
72,
76,
80,
84,
88,
92,
96,
10,
15,
20,
25,
30,
35,
40,
45,
50,
55,
60,
65,
70,
75,
80,
85,
90,
95,
12,
18,
24,
30,
36,
42,
48,
54,
60,
66,
72,
78,
84,
90,
96,
14,
21,
```

```
     28,
     35,
     42,
     49,
     56,
     63,
     70,
     77,
     84,
     91,
     98,
     16,
     24,
     32,
     40,
     48,
     56,
     64,
     72,
     80,
     88,
     96,
     18,
     27,
     36,
     45,
     54,
     63,
     72,
     81,
     90,
     99,
     20,
     30,
     40,
     50,
     60,
     70,
     80,
     90]
```

If we have a look at the content of no_primes, we can see that we have a problem. There are lots of double entries contained in this list:

Output:
```
[4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
 38,
 40,
 42,
 44,
 46,
 48,
 50,
 52,
 54,
 56,
 58,
 60,
 62,
 64,
 66,
 68,
 70,
 72,
 74,
 76,
 78,
 80,
 82,
 84,
 86,
 88,
 90,
 92,
 94,
```

96,
98,
6,
9,
12,
15,
18,
21,
24,
27,
30,
33,
36,
39,
42,
45,
48,
51,
54,
57,
60,
63,
66,
69,
72,
75,
78,
81,
84,
87,
90,
93,
96,
99,
8,
12,
16,
20,
24,
28,
32,
36,
40,
44,
48,
52,

56,
60,
64,
68,
72,
76,
80,
84,
88,
92,
96,
10,
15,
20,
25,
30,
35,
40,
45,
50,
55,
60,
65,
70,
75,
80,
85,
90,
95,
12,
18,
24,
30,
36,
42,
48,
54,
60,
66,
72,
78,
84,
90,
96,
14,
21,

```
    28,
    35,
    42,
    49,
    56,
    63,
    70,
    77,
    84,
    91,
    98,
    16,
    24,
    32,
    40,
    48,
    56,
    64,
    72,
    80,
    88,
    96,
    18,
    27,
    36,
    45,
    54,
    63,
    72,
    81,
    90,
    99,
    20,
    30,
    40,
    50,
    60,
    70,
    80,
    90]
```

The solution to this unbearable problem is easier than you may think. It's just a matter of changing square brackets into braces, or in other words: We will use set comprehension.

SET COMPREHENSION

A set comprehension is similar to a list comprehension, but returns a set and not a list. Syntactically, we use curly brackets instead of square brackets to create a set. Set comprehension is the right functionality to solve our problem from the previous subsection. We are able to create the set of non primes without doublets:

```python
from math import sqrt
n = 100
sqrt_n = int(sqrt(n))
no_primes = {j for i in range(2, sqrt_n+1) for j in range(i*2, n,
i)}
no_primes
```

{4,
6,
8,
9,
10,
12,
14,
15,
16,
18,
20,
21,
22,
24,
25,
26,
27,
28,
30,
32,
33,
34,
35,
36,
38,
39,
40,
42,
44,
45,
46,
48,
49,
50,
51,
52,
54,
55,
56,
57,
58,
60,
62,
63,
64,
65,

```
66,
68,
69,
70,
72,
74,
75,
76,
77,
78,
80,
81,
82,
84,
85,
86,
87,
88,
90,
91,
92,
93,
94,
95,
96,
98,
99}
```

```python
primes = {i for i in range(2, n) if i not in no_primes}
print(primes)
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
1, 67, 71, 73, 79, 83, 89, 97}
```

RECURSIVE FUNCTION TO CALCULATE THE PRIMES

The following Python script uses a recursive function to calculate the prime numbers. It incorporates the fact that it is enough to examine the multiples of the prime numbers up to the square root of n:

```python
from math import sqrt
def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return []
    else:
```

```
        p = primes(int(sqrt(n)))
        no_p = {j for i in p for j in range(i*2, n+1, i)}
        p = {x for x in range(2, n + 1) if x not in no_p}
    return p

for i in range(1,50):
    print(i, primes(i))
```

```
1 []
2 {2}
3 {2, 3}
4 {2, 3}
5 {2, 3, 5}
6 {2, 3, 5}
7 {2, 3, 5, 7}
8 {2, 3, 5, 7}
9 {2, 3, 5, 7}
10 {2, 3, 5, 7}
11 {2, 3, 5, 7, 11}
12 {2, 3, 5, 7, 11}
13 {2, 3, 5, 7, 11, 13}
14 {2, 3, 5, 7, 11, 13}
15 {2, 3, 5, 7, 11, 13}
16 {2, 3, 5, 7, 11, 13}
17 {2, 3, 5, 7, 11, 13, 17}
18 {2, 3, 5, 7, 11, 13, 17}
19 {2, 3, 5, 7, 11, 13, 17, 19}
20 {2, 3, 5, 7, 11, 13, 17, 19}
21 {2, 3, 5, 7, 11, 13, 17, 19}
22 {2, 3, 5, 7, 11, 13, 17, 19}
23 {2, 3, 5, 7, 11, 13, 17, 19, 23}
24 {2, 3, 5, 7, 11, 13, 17, 19, 23}
25 {2, 3, 5, 7, 11, 13, 17, 19, 23}
26 {2, 3, 5, 7, 11, 13, 17, 19, 23}
27 {2, 3, 5, 7, 11, 13, 17, 19, 23}
28 {2, 3, 5, 7, 11, 13, 17, 19, 23}
29 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
30 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
31 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
32 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
33 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
34 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
35 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
36 {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
37 {2, 3, 5, 37, 7, 11, 13, 17, 19, 23, 29, 31}
38 {2, 3, 5, 37, 7, 11, 13, 17, 19, 23, 29, 31}
39 {2, 3, 5, 37, 7, 11, 13, 17, 19, 23, 29, 31}
40 {2, 3, 5, 37, 7, 11, 13, 17, 19, 23, 29, 31}
41 {2, 3, 5, 37, 7, 41, 11, 13, 17, 19, 23, 29, 31}
42 {2, 3, 5, 37, 7, 41, 11, 13, 17, 19, 23, 29, 31}
43 {2, 3, 5, 37, 7, 41, 11, 43, 13, 17, 19, 23, 29, 31}
44 {2, 3, 5, 37, 7, 41, 11, 43, 13, 17, 19, 23, 29, 31}
45 {2, 3, 5, 37, 7, 41, 11, 43, 13, 17, 19, 23, 29, 31}
46 {2, 3, 5, 37, 7, 41, 11, 43, 13, 17, 19, 23, 29, 31}
```

```
47 {2, 3, 5, 37, 7, 41, 11, 43, 13, 47, 17, 19, 23, 29, 31}
48 {2, 3, 5, 37, 7, 41, 11, 43, 13, 47, 17, 19, 23, 29, 31}
49 {2, 3, 5, 37, 7, 41, 11, 43, 13, 47, 17, 19, 23, 29, 31}
```

DIFFERENCES BETWEEN VERSION 2.X AND 3.X

In Python 2, the loop control variable is not local, i.e. it can change another variable of that name outside of the list comprehension, as we can see in the following example:

```
x = "This value will be changed in the list comprehension"
res = [x for x in range(3)]
res
[0, 1, 2]
x
2
res = [i for i in range(5)]
i
4
```

Guido van Rossum referred to this effect as "one of Python's 'dirty little secrets' for years".1 The reason for doing this was efficiency. "It started out as an intentional compromise to make list comprehensions blindingly fast, and while it was not a common pitfall for beginners, it definitely stung people occasionally."2

This "dirty little secret" is fixed in Python3, as you can see in the following code:

```
x = "Python 3 fixed the dirty little secret"
res = [x for x in range(3)]
print(res)
[0, 1, 2]
x
```

```
[0, 1, 2]
```

Output: `'Python 3 fixed the dirty little secret'`

Footnotes: 1 Guido van Rossum: From List Comprehensions to Generator Expressions

2 dto.

GENERATORS AND ITERATORS

INTRODUCTION

What is an iterator? Iterators are objects that can be iterated over like we do in a for loop. We can also say that an iterator is an object, which returns data, one element at a time. That is, they do not do any work until we explicitly ask for their next item. They work on a principle, which is known in computer science as *lazy evaluation*. Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is really needed. Due to the laziness of Python iterators, they are a great way to deal with infinity, i.e. iterables which can iterate for ever. You can hardly find Python programs that are not teaming with iterators.

Iterators are a fundamental concept of Python. You already learned in your first Python programs that you can iterate over container objects such as lists and strings. To do this, Python creates an iterator version of the list or string. In this case, an iterator can be seen as a pointer to a container, which enables us to iterate over all the elements of this container. An iterator is an abstraction, which enables the programmer to access all the elements of an iterable object (a set, a string, a list etc.) without any deeper knowledge of the data structure of this object.

Generators are a special kind of function, which enable us to implement or generate iterators.

Mostly, iterators are implicitly used, like in the for-loop of Python. We demonstrate this in the following example. We are iterating over a list, but you shouldn't be mistaken: A list is not an iterator, but it can be used like an iterator:

```python
cities = ["Paris", "Berlin", "Hamburg",
          "Frankfurt", "London", "Vienna",
          "Amsterdam", "Den Haag"]
for location in cities:
    print("location: " + location)
```

```
location: Paris
location: Berlin
location: Hamburg
location: Frankfurt
location: London
location: Vienna
location: Amsterdam
location: Den Haag
```

What is really going on when a for loop is executed? The function 'iter' is applied to the object following the 'in' keyword, e.g. for i in o:. Two cases are possible: o is either iterable or not. If o is not iterable, an exception will be raised, saying that the type of the object is not iterable. On the other hand, if o is iterable the call iter(o) will return an iterator, let us call it iterator_obj The for loop uses this iterator to iterate over the object o by using the next method. The for loop stops when next(iterator_obj) is exhausted, which means it returns a StopIteration exception. We demonstrate this behaviour in the following code example:

```python
expertises = ["Python Beginner",
              "Python Intermediate",
              "Python Proficient",
              "Python Advanced"]
expertises_iterator = iter(expertises)
print("Calling 'next' for the first time: ", next(expertises_itera
tor))
print("Calling 'next' for the second time: ", next(expertises_iter
ator))
```

```
Calling 'next' for the first time:  Python Beginner
Calling 'next' for the second time:  Python Intermediate
```

We could have called `next` two more times, but after this we will get a StopIteration Exception.

We can simulate this iteration behavior of the for loop in a while loop: You might have noticed that there is something missing in our program: We have to catch the "Stop Iteration" exception:

```python
other_cities = ["Strasbourg", "Freiburg", "Stuttgart",
                "Vienna / Wien", "Hannover", "Berlin",
                "Zurich"]

city_iterator = iter(other_cities)
while city_iterator:
    try:
        city = next(city_iterator)
        print(city)
    except StopIteration:
        break
```

```
Strasbourg
Freiburg
Stuttgart
Vienna / Wien
Hannover
Berlin
Zurich
```

The sequential base types as well as the majority of the classes of the standard library of Python support iteration. The dictionary data type dict supports iterators as well. In this case the iteration runs over the keys of the dictionary:

```python
capitals = {
    "France":"Paris",
    "Netherlands":"Amsterdam",
    "Germany":"Berlin",
    "Switzerland":"Bern",
    "Austria":"Vienna"}

for country in capitals:
    print("The capital city of " + country + " is " + capitals[co
untry])
```

```
The capital city of France is Paris
The capital city of Netherlands is Amsterdam
The capital city of Germany is Berlin
The capital city of Switzerland is Bern
The capital city of Austria is Vienna
```

Off-topic: Some readers may be confused to learn from our example that the capital of the Netherlands is not Den Haag (The Hague) but Amsterdam. Amsterdam is the capital of the Netherlands according to the constitution, even though the Dutch parliament and the Dutch government are situated in The Hague, as well as the Supreme Court and the Council of State.

IMPLEMENTING AN ITERATOR AS A CLASS

One way to create iterators in Python is defining a class which implements the methods `__init__` and `__next__`. We show this by implementing a class cycle, which can be used to cycle over an iterable object forever. In other words, an instance of this class returns the element of an iterable until it is exhausted. Then it repeats the sequence indefinitely.

```python
class Cycle(object):

    def __init__(self, iterable):
        self.iterable = iterable
        self.iter_obj = iter(iterable)

    def __iter__(self):
        return self

    def __next__(self):
        while True:
```

```python
            try:
                next_obj = next(self.iter_obj)
                return next_obj
            except StopIteration:
                self.iter_obj = iter(self.iterable)


x = Cycle("abc")

for i in range(10):
    print(next(x), end=", ")
```

a, b, c, a, b, c, a, b, c, a,

Even though the object-oriented approach to creating an iterator may be very interesting, this is not the pythonic method.

The usual and easiest way to create an iterator in Python consists in using a generator function. You will learn this in the following chapter.

GENERATORS

On the surface, generators in Python look like functions, but there is both a syntactic and a semantic difference. One distinguishing characteristic is the *yield* statements. The yield statement turns a functions into a generator. A generator is a function which returns a generator object. This generator object can be seen like a function which produces a sequence of results instead of a single object. This sequence of values is produced by iterating over it, e.g. with a for loop. The values, on which can be iterated, are created by using the yield statement. The value created by the yield statement is the value following the yield keyword. The execution of the code stops when a yield statement is reached. The value behind the yield will be returned. The execution of the generator is interrupted now. As soon as "next" is called again on the generator object, the generator function will resume execution right after the yield statement in the code, where the last call is made. The execution will continue in the state in which the generator was left after the last yield. In other words, all the local variables still exist, because they are automatically saved between calls. This is a fundamental difference to functions: functions always start their execution at the beginning of the function body, regardless of where they had left in previous calls. They don't have any static or persistent values. There may be more than one yield statement in the code of a generator or the yield statement might be inside the body of a loop. If there is a return statement in the code of a generator, the execution will stop with a StopIteration exception error when this code is executed by the Python interpreter. The word "generator" is sometimes ambiguously used to mean both the generator function itself and the objects which are generated by a generator.

Everything which can be done with a generator can also be implemented with a class based iterator as well. However, the crucial advantage of generators consists in automatically creating the methods `__iter__` () and next(). Generators provide a very neat way of producing data which is huge or even infinite.

The following is a simple example of a generator, which is capable of producing various city names.

It's possible to create a generator object with this generator, which generates all the city names, one after the other.

```python
def city_generator():
    yield("Hamburg")
    yield("Konstanz")
    yield("Berlin")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

We created an iterator by calling city_generator():

```python
city = city_generator()
```

```python
print(next(city))
```

Hamburg

```python
print(next(city))
```

Konstanz

```python
print(next(city))
```

Berlin

```python
print(next(city))
```

Zurich

```python
print(next(city))
```

Schaffhausen

```python
print(next(city))
```

Stuttgart

```python
print(next(city))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-22-02870f953c83> in <module>
----> 1 print(next(city))

StopIteration:
```

As we can see, we have generated an iterator `city` in the interactive shell. Every call of the method `next(city)` returns another city. After the last city, i.e. Stuttgart, has been created, another call of `next(city)` raises an exception, saying that the iteration has stopped, i.e. `StopIteration`. "Can we send a reset to an iterator?" is a frequently asked question, so that it can start the iteration all over again. There is no reset, but it's possible to create another generator. This can be done e.g. by using the statement "city = city_generator()" again. Though at the first sight the yield statement looks like the return statement of a function, we can see in this example that there is a huge difference. If we had a return statement instead of a yield in the previous example, it would be a function. But this function would always **return** the first city "Hamburg" and never any of the other cities, i.e. "Konstanz", "Berlin", "Zurich", "Schaffhausen", and "Stuttgart"

METHOD OF OPERATION

As we have elaborated in the introduction of this chapter, the generators offer a comfortable method to generate iterators, and that's why they are called generators.

Method of working:

- A generator is called like a function. Its return value is an iterator, i.e. a generator object. The code of the generator will not be executed at this stage.
- The iterator can be used by calling the next method. The first time the execution starts like a function, i.e. the first line of code within the body of the iterator. The code is executed until a yield statement is reached.
- **yield** returns the value of the expression, which is following the keyword yield. This is like a function, but Python keeps track of the position of this yield and the state of the local variables is stored for the next call. At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call.
- The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value.

We will illustrate this behaviour in the following example. The generator count creates an iterator which creates a sequence of values by counting from the start value 'firstval' and using 'step' as the increment for counting:

```
def count(firstval=0, step=1):
    x = firstval
    while True:
        yield x
        x += step

counter = count() # count will start with 0
for i in range(10):
    print(next(counter), end=", ")

start_value = 2.1
stop_value = 0.3
```

```
print("\nNew counter:")
counter = count(start_value, stop_value)
for i in range(10):
    new_value = next(counter)
    print(f"{new_value:2.2f}", end=", ")
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
New counter:
2.10, 2.40, 2.70, 3.00, 3.30, 3.60, 3.90, 4.20, 4.50, 4.80,
```

Fibonacci as a Generator:

The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of filius Bonacci, "son of Bonaccio"). In his textbook Liber Abaci, which appeared in the year 1202) he had an exercise about the rabbits and their breeding: It starts with a newly-born pair of rabbits, i.e. a male and a female. It takes one month until they can mate. At the end of the second month the female gives birth to a new pair of rabbits. Now let's suppose that every female rabbit will bring forth another pair of rabbits every month after the end of the first month. We have to mention that Fibonacci's rabbits never die. They question is how large the population will be after a certain period of time.

This produces a sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13

This sequence can be defined in mathematical terms like this:

$F_n = F_{n-1} + F_{n-2}$ with the seed values: $F_0 = 0$ and $F_1 = 1$

```
def fibonacci(n):
    """ A generator for creating the Fibonacci numbers """
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5)
for x in f:
    print(x, " ", end="")  #
print()
```

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

The generator above can be used to create the first n Fibonacci numbers separated by blanks, or better (n+1) numbers because the 0th number is also included. In the next example we present a version which is capable of returning an endless iterator. We have to take care when we use this iterator that a termination criterion is

used:

```python
def fibonacci():
    """Generates an infinite sequence of Fibonacci numbers on demand"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

f = fibonacci()

counter = 0
for x in f:
    print(x, " ", end="")
    counter += 1
    if (counter > 10):
        break
print()
```

```
0  1  1  2  3  5  8  13  21  34  55
```

USING A 'RETURN' IN A GENERATOR

Since Python 3.3, generators can also use return statements, but a generator still needs at least one yield statement to be a generator! A return statement inside of a generator is equivalent to raise StopIteration()

Let's have a look at a generator in which we raise StopIteration:

```python
def gen():
    yield 1
    raise StopIteration(42)
    yield 2

g = gen()

next(g)
```

Output: 1

```python
next(g)
```

```
---------------------------------------------------------------
StopIteration                              Traceback (most recent call last)
<ipython-input-26-6735830131e8> in gen()
      2         yield 1
----> 3         raise StopIteration(42)
      4         yield 2

StopIteration: 42

The above exception was the direct cause of the following exception:

RuntimeError                               Traceback (most recent call last)
<ipython-input-29-e734f8aca5ac> in <module>
----> 1 next(g)

RuntimeError: generator raised StopIteration
```

We demonstrate now that `return` is "nearly" equivalent to raising the 'StopIteration' exception.

```python
def gen():
    yield 1
    return 42
    yield 2
```

```python
g = gen()
next(g)
```

Output: 1

```python
next(g)
```

```
---------------------------------------------------------------
StopIteration                              Traceback (most recent call last)
<ipython-input-32-e734f8aca5ac> in <module>
----> 1 next(g)

StopIteration: 42
```

SEND METHOD /COROUTINES

Generators can not only send objects but also receive objects. Sending a message, i.e. an object, into the generator can be achieved by applying the send method to the generator object. Be aware of the fact that send both sends a value to the generator and returns the value yielded by the generator. We will demonstrate this behavior in the following simple example of a coroutine:

```python
def simple_coroutine():
```

```python
        print("coroutine has been started!")
    while True:
        x = yield "foo"
        print("coroutine received: ", x)


cr = simple_coroutine()
cr
```

`<generator object simple_coroutine at 0x7fa2a8049ac0>`

```python
next(cr)
```

```
coroutine has been started!
```

`'foo'`

```python
ret_value = cr.send("Hi")
print("'send' returned: ", ret_value)
```

```
coroutine received:  Hi
'send' returned:  foo
```

We had to call next on the generator first, because the generator needed to be started. Using send to a generator which hasn't been started leads to an exception.

To use the send method, the generator must wait for a yield statement so that the data sent can be processed or assigned to the variable on the left. What we haven't said so far: A next call also sends and receives. It always sends a None object. The values sent by "next" and "send" are assigned to a variable within the generator: this variable is called `new_counter_val` in the following example.

The following example modifies the generator 'count' from the previous subchapter by adding a `send` feature.

In [ ]:
```python
def count(firstval=0, step=1):
    counter = firstval
    while True:
        new_counter_val = yield counter
        if new_counter_val is None:
            counter += step
        else:
            counter = new_counter_val

start_value = 2.1
stop_value = 0.3
```

```python
counter = count(start_value, stop_value)
for i in range(10):
    new_value = next(counter)
    print(f"{new_value:2.2f}", end=", ")

print("set current count value to another value:")
counter.send(100.5)
for i in range(10):
    new_value = next(counter)
    print(f"{new_value:2.2f}", end=", ")
```

```python
from random import choice

def song_generator(song_list):
    new_song = None
    while True:
        if new_song != None:
            if new_song not in song_list:
                song_list.append(new_song)
            new_song = yield new_song
        else:
            new_song = yield(choice(song_list))

songs = ["Her Şeyi Yak - Sezen Aksu",
        "Bluesette - Toots Thielemans",
        "Six Marimbas - Steve Reich",
        "Riverside - Agnes Obel",
        "Not for Radio - Nas",
        "What's going on - Taste",
        "On Stream - Nils Petter Molvær",
        "La' Inta Habibi - Fayrouz",
        "Ik Leef Niet Meer Voor Jou - Marco Borsato",
        "Δέκα λεπτά - Αθηνά Ανδρεάδη"]
```

```python
radio_program = song_generator(songs)
```

```python
next(radio_program)
```

Output: `'Riverside - Agnes Obel'`

```python
for i in range(3):
    print(next(radio_program))
```

```
Bluesette - Toots Thielemans
Six Marimbas - Steve Reich
Six Marimbas - Steve Reich
```

```
radio_program.send("Distorted Angels - Archive")
```

Output: `'Distorted Angels - Archive'`

```
songs
```

Output:
```
['Her Şeyi Yak - Sezen Aksu',
 'Bluesette - Toots Thielemans',
 'Six Marimbas - Steve Reich',
 'Riverside - Agnes Obel',
 'Not for Radio - Nas',
 "What's going on - Taste",
 'On Stream - Nils Petter Molvær',
 "La' Inta Habibi - Fayrouz",
 'Ik Leef Niet Meer Voor Jou - Marco Borsato',
 'Δέκα λεπτά - Αθηνά Ανδρεάδη',
 'Distorted Angels - Archive']
```

So far, we could change the behavior of the song interator by sending a new song, i.e. a title plus the performer. We will extend the generator now. We make it possible to send a new song list. We have to send a tuple to the new iterator, either a tuple:

(,) or

("-songlist-",)

```python
from random import choice
def song_generator(song_list):
    new_song = None
    while True:
        if new_song != None:
            if new_song[0] == "-songlist-":
                song_list = new_song[1]
                new_song = yield(choice(song_list))
            else:
                title, performer = new_song
                new_song = title + " - " + performer
                if new_song not in song_list:
                    song_list.append(new_song)
                new_song = yield new_song
        else:
            new_song = yield(choice(song_list))
```

```
songs1 = ["Après un Rêve - Gabriel Fauré"
          "On Stream - Nils Petter Molvær",
          "Der Wanderer Michael - Michael Wollny",
          "Les barricades mystérieuses - Barbara Thompson",
          "Monday - Ludovico Einaudi"]

songs2 = ["Dünyadan Uszak - Pinhani",
           "Again - Archive",
           "If I had a Hear - Fever Ray"
           "Every you, every me - Placebo",
           "Familiar - Angnes Obel"]
```

```
radio_prog = song_generator(songs1)
```

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

```
Les barricades mystérieuses - Barbara Thompson
Après un Rêve - Gabriel FauréOn Stream - Nils Petter Molvær
Les barricades mystérieuses - Barbara Thompson
Les barricades mystérieuses - Barbara Thompson
Après un Rêve - Gabriel FauréOn Stream - Nils Petter Molvær
```

We change the radio program now by exchanging the song list:

```
radio_prog.send(("-songlist-", songs2))
```
Output: `'Familiar - Angnes Obel'`

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

```
Again - Archive
If I had a Hear - Fever RayEvery you, every me - Placebo
Dünyadan Uszak - Pinhani
Dünyadan Uszak - Pinhani
If I had a Hear - Fever RayEvery you, every me - Placebo
```

THE THROW METHOD

The throw() method raises an exception at the point where the generator was paused, and returns the next value yielded by the generator. It raises StopIteration if the generator exits without yielding another value. The generator has to catch the passed-in exception, otherwise the exception will be propagated to the caller.

The infinite loop of our `count` generator from our previous example keeps yielding the elements of the sequential data, but we don't have any information about the index or the state of the variables of "counter".

We can get this information, i.e. the state of the variables in `count` by throwing an exception with the `throw` method. We catch this exception inside of the generator and print the state of the values of "count":

```python
def count(firstval=0, step=1):
    counter = firstval
    while True:
        try:
            new_counter_val = yield counter
            if new_counter_val is None:
                counter += step
            else:
                counter = new_counter_val
        except Exception:
            yield (firstval, step, counter)
```

In the following code block, we will show how to use this generator:

```python
c = count()
for i in range(6):
    print(next(c))
print("Let us see what the state of the iterator is:")
state_of_count = c.throw(Exception)
print(state_of_count)
print("now, we can continue:")
for i in range(3):
    print(next(c))
```

```
0
1
2
3
4
5
Let us see what the state of the iterator is:
(0, 1, 5)
now, we can continue:
5
6
7
```

We can improve the previous example by defining our own exception class StateOfGenerator:

```python
class StateOfGenerator(Exception):
    def __init__(self, message=None):
```

```
            self.message = message

def count(firstval=0, step=1):
    counter = firstval
    while True:
        try:
            new_counter_val = yield counter
            if new_counter_val is None:
                counter += step
            else:
                counter = new_counter_val
        except StateOfGenerator:
            yield (firstval, step, counter)
```

We can use the previous generator like this:

```
c = count()
for i in range(3):
    print(next(c))
print("Let us see what the state of the iterator is:")
i = c.throw(StateOfGenerator)
print(i)
print("now, we can continue:")
for i in range(3):
    print(next(c))
```

```
0
1
2
Let us see what the state of the iterator is:
(0, 1, 2)
now, we can continue:
2
3
4
```

YIELD FROM

"yield from" is available since Python 3.3! The `yield from <expr>` statement can be used inside the body of a generator. `<expr>` has to be an expression evaluating to an iterable, from which an iterator will be extracted. The iterator is run to exhaustion, i.e. until it encounters a `StopIteration` exception. This iterator yields and receives values to or from the caller of the generator, i.e. the one which contains the yield from statement.

We can learn from the following example by looking at the two generators 'gen1' and 'gen2' that the `yield`

`from` from `gen2` are substituting the for loops of 'gen1':

```python
def gen1():
    for char in "Python":
        yield char
    for i in range(5):
        yield i

def gen2():
    yield from "Python"
    yield from range(5)

g1 = gen1()
g2 = gen2()
print("g1: ", end=", ")
for x in g1:
    print(x, end=", ")
print("\ng2: ", end=", ")
for x in g2:
    print(x, end=", ")
print()
```

```
g1: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
g2: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
```

We can see from the output that both generators are the same.

The benefit of a yield from statement can be seen as a way to split a generator into multiple generators. That's what we have done in our previous example and we will demonstrate this more explicitely in the following example:

```python
def cities():
    for city in ["Berlin", "Hamburg", "Munich", "Freiburg"]:
        yield city

def squares():
    for number in range(10):
        yield number ** 2

def generator_all_in_one():
    for city in cities():
        yield city
    for number in squares():
        yield number
```

```
def generator_splitted():
    yield from cities()
    yield from squares()

lst1 = [el for el in generator_all_in_one()]
lst2 = [el for el in generator_splitted()]
print(lst1 == lst2)
```

```
True
```

The previous code returns True because the generators `generator_all_in_one` and `generator_splitted yield` the same elements. This means that if the `<expr>` from the `yield from` is another generator, the effect is the same as if the body of the sub-generator were inlined at the point of the `yield from` statement. Furthermore, the subgenerator is allowed to execute a `return` statement with a value, and that value becomes the value of the `yield from` expression. We demonstrate this with the following little script:

```
def subgenerator():
    yield 1
    return 42

def delegating_generator():
    x = yield from subgenerator()
    print(x)

for x in delegating_generator():
    print(x)
```

```
1
42
```

The full semantics of the yield from expression is described in six points in "PEP 380 -- Syntax for Delegating to a Subgenerator" in terms of the generator protocol:

- Any values that the iterator yields are passed directly to the caller.
- Any values sent to the delegating generator using send() are passed directly to the iterator. If the sent value is None, the iterator's **next**() method is called. If the sent value is not None, the iterator's send() method is called. If the call raises StopIteration, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
- Exceptions other than GeneratorExit thrown into the delegating generator are passed to the throw() method of the iterator. If the call raises StopIteration, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
- If a GeneratorExit exception is thrown into the delegating generator, or the close() method of the delegating generator is called, then the close() method of the iterator is called if it has one. If this

call results in an exception, it is propagated to the delegating generator. Otherwise, GeneratorExit is raised in the delegating generator.
- The value of the yield from expression is the first argument to the StopIteration exception raised by the iterator when it terminates.
- return expr in a generator causes StopIteration(expr) to be raised upon exit from the generator.

RECURSIVE GENERATORS

The following example is a generator to create all the permutations of a given list of items.

For those who don't know what permutations are, we have a short introduction:

Formal Definition: The term "permutation" stems from the latin verb "permutare" which means A permutation is a rearrangement of the elements of an ordered list. In other words: Every arrangement of n elements is called a permutation.

In the following lines we show you all the permutations of the letter a, b and c:

a b c
a c b
b a c
b c a
c a b
c b a

The number of permutations on a set of n elements is given by n!

n! = n$(n-1)$(n-2) ... 2 * 1

n! is called the factorial of n.

The permutation generator can be called with an arbitrary list of objects. The iterator returned by this generator generates all the possible permutations:

```python
def permutations(items):
    n = len(items)
    if n==0: yield []
    else:
        for i in range(len(items)):
            for cc in permutations(items[:i]+items[i+1:]):
                yield [items[i]]+cc

for p in permutations(['r','e','d']): print(''.join(p))
for p in permutations(list("game")): print(''.join(p) + ", ", end="")
```

```
red
rde
erd
edr
dre
der
game, gaem, gmae, gmea, geam, gema, agme, agem, amge, ameg, aegm,
aemg, mgae, mgea, mage, maeg, mega, meag, egam, egma, eagm, eamg,
emga, emag,
```

The previous example can be hard to understand for newbies. Like always, Python offers a convenient solution. We need the module **itertools** for this purpose. Itertools is a very handy tool to create and operate on iterators.

Creating permutations with itertools:

```python
import itertools
perms = itertools.permutations(['r','e','d'])
list(perms)
```

Output:
```
[('r', 'e', 'd'),
 ('r', 'd', 'e'),
 ('e', 'r', 'd'),
 ('e', 'd', 'r'),
 ('d', 'r', 'e'),
 ('d', 'e', 'r')]
```

The term "permutations" can sometimes be used in a weaker meaning. Permutations can denote in this weaker meaning a sequence of elements, where each element occurs just once, but without the requirement to contain all the elements of a given set. So in this sense (1,3,5,2) is a permutation of the set of digits {1,2,3,4,5,6}. We can build, for example, all the sequences of a fixed length k of elements taken from a given set of size n with k ≤ n.

These are are all the 3-permutations of the set {"a","b","c","d"}.

These atypical permutations are also known as sequences without repetition. By using this term we can avoid confusion with the term "permutation". The number of such k-permutations of n is denoted by $P_{n,k}$ and its value is calculated by the product:

$n \cdot (n-1) \cdot \ldots (n-k+1)$

By using the factorial notation, the above mentioned expression can be written as:

$P_{n,k} = n!/(n-k)!$

A generator for the creation of k-permuations of n objects looks very similar to our previous permutations generator:

```python
def k_permutations(items, n):
    if n==0:
        yield []
    else:
        for item in items:
            for kp in k_permutations(items, n-1):
                if item not in kp:
                    yield [item] + kp

for kp in k_permutations("abcd", 3):
    print(kp)
```

```
['a', 'b', 'c']
['a', 'b', 'd']
['a', 'c', 'b']
['a', 'c', 'd']
['a', 'd', 'b']
['a', 'd', 'c']
['b', 'a', 'c']
['b', 'a', 'd']
['b', 'c', 'a']
['b', 'c', 'd']
['b', 'd', 'a']
['b', 'd', 'c']
['c', 'a', 'b']
['c', 'a', 'd']
['c', 'b', 'a']
['c', 'b', 'd']
['c', 'd', 'a']
['c', 'd', 'b']
['d', 'a', 'b']
['d', 'a', 'c']
['d', 'b', 'a']
['d', 'b', 'c']
['d', 'c', 'a']
['d', 'c', 'b']
```

A GENERATOR OF GENERATORS

The second generator of our Fibonacci sequence example generates an iterator, which can theoretically produce all the Fibonacci numbers, i.e. an infinite number. But you shouldn't try to produce all these numbers in a list with the following line.

```
    list(fibonacci())
```

This will show you very fast the limits of your computer. In most practical applications, we only need the first `n` elements of and other "endless" iterators. We can use another generator, in our example `firstn`, to create the first `n` elements of a generator `generator`:

```python
def firstn(generator, n):
    g = generator()
    for i in range(n):
        yield next(g)
```

The following script returns the first 10 elements of the Fibonacci sequence:

```python
def fibonacci():
    """ A Fibonacci number generator """
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

print(list(firstn(fibonacci, 10)))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

EXERCISES

EXERCISE 1

Write a generator which computes the running average.

Write a generator `frange`, which behaves like `range` but accepts `float` values.

3) Write a generator `trange`, which generates a sequence of time tuples from start to stop incremented by step. A time tuple is a 3-tuple of integers: (hours, minutes, seconds) So a call to `trange` might look like this:
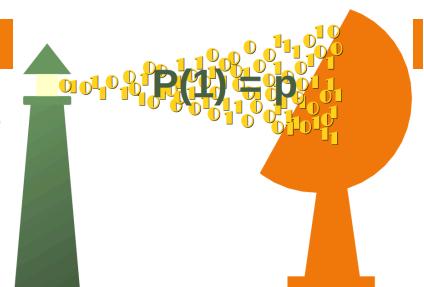
```
trange((10, 10, 10), (13, 50, 15), (0, 15, 12) )
```

Write a version "rtrange" of the previous generator, which can receive messages to reset the start value.

Write a program, using the newly written generator "trange", to create a file "times_and_temperatures.txt". The lines of this file contain a time in the format `hh::mm::ss` and random temperatures between 10.0 and 25.0 degrees. The times should be ascending in steps of 90 seconds starting with 6:00:00. For example:

```
06:00:00 20.1
06:01:30 16.1
06:03:00 16.9
06:04:30 13.4
06:06:00 23.7
06:07:30 23.6
06:09:00 17.5
06:10:30 11.0
```

Write a generator with the name "random_ones_and_zeroes", which returns a bitstream, i.e. a zero or a one in every iteration. The probability p for returning a 1 is defined in a variable p. The generator will initialize this value to 0.5. In other words, zeroes and ones will be returned with the same probability.

We wrote a class `Cycle` in the beginning of this chapter of our Python tutorial. Write a generator `cycle` performing the same task.

SOLUTIONS TO OUR EXERCISES

SOLUTION TO EXERCISE 1

```python
def running_average():
    total = 0.0
    counter = 0
    average = None
    while True:
        term = yield average
```

```
        total += term
        counter += 1
        average = total / counter


ra = running_average()   # initialize the coroutine
next(ra)                 # we have to start the coroutine
for value in [7, 13, 17, 231, 12, 8, 3]:
    out_str = "sent: {val:3d}, new average: {avg:6.2f}"
    print(out_str.format(val=value, avg=ra.send(value)))
```

```
sent:   7, new average:    7.00
sent:  13, new average:   10.00
sent:  17, new average:   12.33
sent: 231, new average:   67.00
sent:  12, new average:   56.00
sent:   8, new average:   48.00
sent:   3, new average:   41.57
```

```
def frange(*args):
    """  dfgdg """
    startval = 0
    stepsize = 1
    if len(args) == 1:
        endval = args[0]
    elif len(args) == 2:
        startval, endval = args
    elif len(args) == 3:
        startval, endval, stepsize = args
    else:
        txt = "range expected at most 3 arguments, got " + len(arg
s)
        raise TypeError(txt)

    value = startval
    factor = -1 if stepsize < 0 else 1
    while (value - endval) * (factor) < 0:
        yield value
        value += stepsize
```

Using `frange` may llok like this:

```
for i in frange(5.6):
    print(i, end=", ")
print()
for i in frange(0.3, 5.6):
    print(i, end=", ")
print()
for i in frange(0.3, 5.6, 0.8):
    print(i, end=", ")
print()
```

```
0, 1, 2, 3, 4, 5,
0.3, 1.3, 2.3, 3.3, 4.3, 5.3,
0.3, 1.1, 1.9000000000000001, 2.7, 3.5, 4.3, 5.1,
```

```python
def trange(start, stop, step):
    """
    trange(stop) -> time as a 3-tuple (hours, minutes, seconds)
    trange(start, stop[, step]) -> time tuple

    start: time tuple (hours, minutes, seconds)
    stop: time tuple
    step: time tuple

    returns a sequence of time tuples from start to stop increment
ed by step
    """

    current = list(start)
    while current < list(stop):
        yield tuple(current)
        seconds = step[2] + current[2]
        min_borrow = 0
        hours_borrow = 0
        if seconds < 60:
            current[2] = seconds
        else:
            current[2] = seconds - 60
            min_borrow = 1
        minutes = step[1] + current[1] + min_borrow
        if minutes < 60:
            current[1] = minutes
        else:
            current[1] = minutes - 60
```

```
            hours_borrow = 1
        hours = step[0] + current[0] + hours_borrow
        if hours < 24:
            current[0] = hours
        else:
            current[0] = hours - 24
```

Overwriting timerange.py

```
from timerange import trange

for time in trange((10, 10, 10), (19, 53, 15), (1, 24, 12) ):
    print(time)
```

```
(10, 10, 10)
(11, 34, 22)
(12, 58, 34)
(14, 22, 46)
(15, 46, 58)
(17, 11, 10)
(18, 35, 22)
```

```
def rtrange(start, stop, step):
        """
        trange(stop) -> time as a 3-tuple (hours, minutes, second
s)
        trange(start, stop[, step]) -> time tuple

        start: time tuple (hours, minutes, seconds)
        stop: time tuple
        step: time tuple

        returns a sequence of time tuples from start to stop incre
mented by step

        The generator can be reset by sending a new "start" value.
        """

        current = list(start)
        while current < list(stop):
            new_start = yield tuple(current)
            if new_start != None:
                current = list(new_start)
```

```
                continue
            seconds = step[2] + current[2]
            min_borrow = 0
            hours_borrow = 0
            if seconds < 60:
                current[2] = seconds
            else:
                current[2] = seconds - 60
                min_borrow = 1
            minutes = step[1] + current[1] + min_borrow
            if minutes < 60:
                current[1] = minutes
            else:
                current[1] = minutes - 60
                hours_borrow = 1
            hours = step[0] + current[0] + hours_borrow
            if hours < 24:
                current[0] = hours
            else:
                current[0] = hours -24
```

Overwriting rtimerange.py

```python
from rtimerange import rtrange


ts = rtrange((10, 10, 10), (17, 50, 15), (1, 15, 12) )
for _ in range(3):
    print(next(ts))

print(ts.send((8, 5, 50)))
for _ in range(3):
    print(next(ts))
```

```
(10, 10, 10)
(11, 25, 22)
(12, 40, 34)
(8, 5, 50)
(9, 21, 2)
(10, 36, 14)
(11, 51, 26)
```

```python
from timerange import trange
```

```python
import random

fh = open("times_and_temperatures.txt", "w")

for time in trange((6, 0, 0), (23, 0, 0), (0, 1, 30) ):
    random_number = random.randint(100, 250) / 10
    lst = time + (random_number,)
    output = "{:02d}:{:02d}:{:02d} {:4.1f}\n".format(*lst)
    fh.write(output)
```

You can find further details and the mathematical background about this exercise in our chapter on <u>Weighted</u> <u>Probabilities</u>

```python
import random

def random_ones_and_zeros():
    p = 0.5
    while True:
        x = random.random()
        message = yield 1 if x < p else 0
        if message != None:
            p = message

x = random_ones_and_zeros()
next(x)  # we are not interested in the return value
for p in [0.2, 0.8]:
    print("\nWe change the probability to : " + str(p))
    x.send(p)
    for i in range(20):
        print(next(x), end=" ")
print()
```

```
We change the probability to : 0.2
1 0 0 0 1 0 0 1 1 0 1 0 1 0 0 0 1 1 0 0
We change the probability to : 0.8
0 1 1 0 1 0 1 1 0 0 1 1 1 1 1 0 1 1 1 1
```

The "cycle" generator is part of the module 'itertools'. The following code is the implementation in itertools:

```python
def cycle(iterable):
```

```python
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element


countries = ["Germany", "Switzerland", "Austria"]
country_iterator = cycle(countries)
for i in range(7):
    print(next(country_iterator))
```
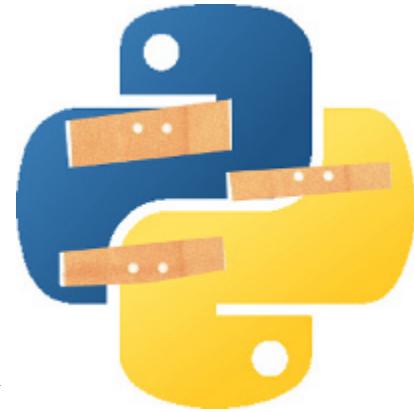
```
Germany
Switzerland
Austria
Germany
Switzerland
Austria
Germany
```

ERRORS AND EXCEPTIONS

EXCEPTION HANDLING

An exception is an error that happens during the execution of a program. Exceptions are known to non-programmers as instances that do not conform to a general rule. The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule". Exception handling is a construct in some programming languages to handle or deal with errors automatically. Many programming languages like C++, Objective-C, PHP, Java, Ruby, Python, and many others have built-in support for exception handling.

Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the programm can be continued afterwards with the previously saved data.

EXCEPTION HANDLING IN PYTHON

Exception handling in Python is very similar to Java. The code, which harbours the risk of an exception, is embedded in a try block. While in Java exceptions are caught by catch clauses, in Python we have statements introduced by an "except" keyword. It's possible to create "custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.

Let's look at a simple example. Assuming we want to ask the user to enter an integer number. If we use a input(), the input will be a string, which we have to cast into an integer. If the input isn't a valid integer, we will generate (raise) a ValueError. We show this in the following interactive session:

In [ ]:

```
n = int(input("Please enter a number: "))
```

With the aid of exception handling, we can write robust code for reading an integer from input:

In [ ]:

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
```

```
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

It's a loop, which breaks only if a valid integer has been given. The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left. If an exception occurs, i.e. in the casting of n, the rest of the try block will be skipped and the except clause will be executed. The raised error, in our case a ValueError, has to match one of the names after except. In our example only one, i.e. "ValueError:". After having printed the text of the print statement, the execution does another loop. It starts with a new input().

We could turn the code above into a function, which can be used to have a foolproof input.

```python
def int_input(prompt):
    while True:
        try:
            age = int(input(prompt))
            return age
        except ValueError as e:
            print("Not a proper integer! Try it again")
```

We use this with our dog age example from the chapter Conditional Statements.

```python
def dog2human_age(dog_age):
    human_age = -1
    if dog_age < 0:
        human_age = -1
    elif dog_age == 0:
        human_age = 0
    elif dog_age == 1:
        human_age = 14
    elif dog_age == 2:
        human_age = 22
    else:
        human_age = 22 + (dog_age -2) * 5
    return human_age
```

```python
age = int_input("Age of your dog? ")
print("Age of the dog: ", dog2human_age(age))
```

```
Not a proper integer! Try it again

Not a proper integer! Try it again

Age of the dog:  37
```

MULTIPLE EXCEPT CLAUSES

A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.

Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:

```
an IOError
ValueError
```

Just in case we have an additional unnamed except clause for an unexpected error:

```python
import sys

try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
    # e can be printed directly without using .args:
    # print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

```
I/O error(2): No such file or directory
```

The handling of the IOError in the previous example is of special interest. The except clause for the IOError specifies a variable "e" after the exception name (IOError). The variable "e" is bound to an exception instance with the arguments stored in instance.args. If we call the above script with a non-existing file, we get the message:

I/O error(2): No such file or directory

And if the file integers.txt is not readable, e.g. if we don't have the permission to read it, we get the following message:

I/O error(13): Permission denied

An except clause may name more than one exception in a tuple of error names, as we see in the following example:

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred")
except:
    print("An unexpected error occurred")
    raise
```

```
An I/O error or a ValueError occurred
```

We want to demonstrate now, what happens, if we call a function within a try block and if an exception occurs inside the function call:

```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
    print("got it :-) ", e)



print("Let's get on")
```

```
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

the function catches the exception.

We will extend our example now so that the function will catch the exception directly:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)

try:
    f()
except ValueError as e:
    print("got it :-) ", e)
```

```
print("Let's get on")
```

```
got it in the function :-)  invalid literal for int() with base 1
0: 'four'
Let's get on
```

As we have expected, the exception is caught inside the function and not in the callers exception:

We add now a "raise", which generates the ValueError again, so that the exception will be propagated to the caller:

```python
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

```
got it in the function :-)  invalid literal for int() with base 1
0: 'four'
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

CUSTOM-MADE EXCEPTIONS

It's possible to create Exceptions yourself:

```python
raise SyntaxError("Sorry, my fault!")
```

```
Traceback (most recent call last):

  File "C:\Users\melis\Anaconda3\lib\site-packages\IPython\core\in
teractiveshell.py", line 3326, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-15-a5649918d59e>", line 1, in <module>
    raise SyntaxError("Sorry, my fault!")

  File "<string>", line unknown
SyntaxError: Sorry, my fault!
```

The best or the Pythonic way to do this, consists in defining an exception class which inherits from the Exception class. You will have to go through the chapter on [Object Oriented Programming](#) to fully understand the following example:

```python
class MyException(Exception):
    pass


raise MyException("An exception doesn't always prove the rule!")
```

```
-----------------------------------------------------------------------------
MyException                               Traceback (most recent call last)
<ipython-input-3-d75bff75fe3a> in <module>
      2         pass
      3
----> 4 raise MyException("An exception doesn't always prove the rule!")

MyException: An exception doesn't always prove the rule!
```

CLEAN-UP ACTIONS (TRY ... FINALLY)

So far the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a **finally** clause. Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not. A simple example to demonstrate the finally clause:

```python
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

```
Your number: 34
There may or may not have been an exception.
The inverse:  0.029411764705882353
```

COMBINING TRY, EXCEPT AND FINALLY

"finally" and "except" can be used together for the same try block, as it can be seen in the following Python example:

```python
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

```
Your number: 23
There may or may not have been an exception.
```

ELSE CLAUSE

The try ... except statement has an optional else clause. An else block has to be positioned after all the except clauses. An else clause will be executed if the try clause doesn't raise an exception.

The following example opens a file and reads in all the lines into a list called "text":

```python
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
    text = fh.readlines()
    fh.close()
except IOError:
    print('cannot open', file_name)

if text:
    print(text[100])
```

```
cannot open -f
```

This example receives the file name via a command line argument. So make sure that you call it properly:

Let's assume that you saved this program as "exception_test.py". In this case, you have to call it with

```
python exception_test.py integers.txt
```

If you don't want this behaviour, just change the line "file_name = sys.argv[1]" to "file_name = 'integers.txt'".

The previous example is nearly the same as:

```python
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    text = fh.readlines()
    fh.close()

if text:
    print(text[100])
```

```
cannot open -f
```

The main difference is that in the first case, all statements of the try block can lead to the same error message "cannot open ...", which is wrong, if fh.close() or fh.readlines() raise an error.

PYTHON TESTS

ERRORS AND TESTS

Usually, programmers and program developers spend a great deal of their time with debugging and testing. It's hard to give exact percentages, because it highly depends on other factors like the individual programming style, the problems to be solved and of course on the qualification of a programmer. Without doubt, the programming language is another important factor.

You don't have to program to get pestered by errors, as even the ancient Romans knew. The philosopher Cicero coined more than 2000 years ago an unforgettable aphorism, which is often quoted: "errare humanum est"* This aphorism is often used as an excuse for failure. Even though it's hardly possible to completely eliminate all errors in a software product, we should always work ambitiously to this end, i.e. to keep the number of errors minimal.

KINDS OF ERRORS

There are various kinds of errors. During program development there are lots of "small errors", mostly typos. Whether a colon is missing - for example, behind an "if" or an "else" - or the keyword "True" is wrongly written with a lower case "t" can make a big difference. These errors are called syntactical errors.** In most cases, syntactical errors can easily be found, but other type of errors are harder to solve. A semantic error is a syntactically correct code, but the program doesn't behave in the intended way. Imagine somebody wants to increment the value of a variable x by one, but instead of "x += 1" he or she writes "x = 1". The following longer code example may harbour another semantic error:

```python
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
else:
    print("So what?")
```

```
So what?
```

We can see two if statements. One nested inside of the other. The code is definitely syntactically correct. But it can be the case that the writer of the program only wanted to output "So what?", if the value of the variable x is both greater than 10 and x is not equal to y. In this case, the code should look like this:

```python
x = int(input("x? "))
y = int(input("y? "))
```

```python
if x > 10:
    if y == x:
        print("Fine")
    else:
        print("So what?")
```

Both code versions are syntactically correct, but one of them violates the intended semantics. Let's look at another example:

```python
for i in range(7):
    print(i)
```

```
0
1
2
3
4
5
6
```

The statement ran without raising an exception, so we know that it is syntactically correct. Though it is not possible to decide if the statement is semantically correct, as we don't know the problem. It may be that the programmer wanted to output the numbers from 1 to 7, i.e. 1,2,...7 In this case, he or she does not properly understand the range function.

So we can divide semantic errors into two categories.

- Errors caused by lack of understanding of a language construct.
- Errors due to logically incorrect code conversion.

UNIT TESTS

This paragraph is about unit tests. As the name implies they are used for testing units or components of the code, typically, classes or functions. The underlying concept is to simplify the testing of large programming systems by testing "small" units. To accomplish this the parts of a program have to be isolated into independent testable "units". One can define "unit testing" as a method whereby individual units of source code are tested to determine if they meet the requirements, i.e. return the expected output for all possible - or defined - input data. A unit can be seen as the smallest testable part of a program, which are often functions or methods from classes. Testing one unit should be independent from the other units as a unit is "quite" small, i.e. manageable to ensure complete correctness. Usually, this is not possible for large scale systems like large software programs or operating systems.

MODULE TESTS WITH NAME

Every module has a name, which is defined in the built-in attribute `__name__`. Let's assume that we have written a module "xyz" which we have saved as "xyz.py". If we import this module with "import xyz", the string "xyz" will be assigned to `__name__`. If we call the file xyz.py as a standalone program, i.e. in the following way,

```
$python3 xyz.py
```

the value of `__name__` will be the string `'__main__'`.

The following module can be used for calculating fibonacci numbers. But it is not important what the module is doing. We want to demonstrate, how it is possible to create a simple module test inside of a module file, - in our case the file "xyz.py", - by using an if statement and checking the value of `__name__`. We check if the module has been started standalone, in which case the value of `__name__` will be `__main__`. Please save the following code as "fibonacci1.py":

-Fibonacci Module-

```python
def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th generat
ion """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

It's possible to test this module manually in the interactive Python shell:

```python
from fibonacci1 import fib, fiblist
fib(0)
```

Test for the fib function was successful!

Output: 0

```python
fib(1)
```

Output: 1

```
fib(10)
```

Output: 55

```
fiblist(10)
```

Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

```
fiblist(-8)
```

Output: [0, 1]

```
fib(-1)
```

Output: 0

```
fib(0.5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-d518fced90e3> in <module>
----> 1 fib(0.5)

~/Dropbox (Bodenseo)/Bodenseo Team Folder/melisa/notebooks_en/fibonacci1.p
y in fib(n)
      2     """ Calculates the n-th Fibonacci number iteratively """
      3     a, b = 0, 1
----> 4     for i in range(n):
      5         a, b = b, a + b
      6     return a

TypeError: 'float' object cannot be interpreted as an integer
```

We can see that the functions make only sense, if the input consists of positive integers. The function fib returns 0 for a negative input and fiblist returns always the list [0,1], if the input is a negative integer. Both functions raise a TypeError exception, because the range function is not defined for floats. We can test our module by checking the return values for some characteristic calls to fib() and fiblist(). So we can add the following if statement

```
if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
    print("Test for the fib function was successful!")
else:
    print("The fib function is returning wrong values!")
```

to our module, but give it a new name `fibonacci2.py` . We can import this module now in a Python shell or inside of a Python program. If the program with the import gets executed, we receive the following output:

```
import fibonacci2
```

```
Test for the fib function was successful!
```

This approach has a crucial disadvantage. If we import the module, we will get output, saying the test was okay. This is omething we don't want to see, when we import the module. Apart from being disturbing it is not common practice. Modules should be silent when being imported, i.e. modules should not produce any output. We can prevent this from happening by using the special built-in variable `__name__`. We can guard the test code by putting it inside the following if statement:

```
if __name__ == "__main__":
    if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
        print("Test for the fib function was successful!")
    else:
        print("The fib function is returning wrong values!")
```

The value of the variable `__name__` is set automatically by Python. Let us imagine that we import some crazy module with the names foobar.py blubla.py and blimblam.py, the values of the variable `__name__` will be `foobar`, `blubla` and `blimblam` correspondingly.

If we change our fibonacci module correspondingly and save it as fibonacci3.py, we get a silent import:

```
import fibonacci3
```

We were successful at silencing the output. Yet, our module should perform the test, if it is started standalone.

```
(base) bernd@moon:~/$ python fibonacci3.py
Test for the fib function was successful!
(base) bernd@moon:~/$
```

If you want to start a Python program from inside of another Python program, you can do this by using the `exec` command, as we do in the following code:

```
exec(open("fibonacci3.py").read())
```

```
Test for the fib function was successful!
```

We will deliberately add an error into our code now.

We change the following line

```
a, b = 0, 1
```

into

```
a, b = 1, 1
```

and save as fibonacci4.py.

Principally, the function fib is still calculating the Fibonacci values, but fib(n) is returning the Fibonacci value for the argument "n+1". If we call our changed module, we receive this error message:

```
exec(open("fibonacci4.py").read())
```

```
The fib function is returning wrong values!
```

Let's rewrite our module:

```python
""" Fibonacci Module """
def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th generat
ion """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib

if __name__ == "__main__":
    if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
        print("Test for the fib function was successful!")
    else:
        print("The fib function is returning wrong values!")
```

```
Test for the fib function was successful!
```

```python
import fibonacci5
```

We have squelched our module now. There will be no messages, if the module is imported. This is the simplest and widest used method for unit tests. But it is definitely not the best one.

DOCTEST MODULE

The doctest module is often considered easier to use than the unittest, though the latter is more suitable for more complex tests. doctest is a test framework that comes prepackaged with Python. The doctest module

searches for pieces of text that look like interactive Python sessions inside the documentation parts of a module, and then executes (or reexecutes) the commands of those sessions to verify that they work exactly as shown, i.e. that the same results can be achieved. In other words: The help text of the module is parsed, for example, python sessions. These examples are run and the results are compared to the expected value.

Usage of doctest: "doctest" has to be imported. The part of an interactive Python sessions with the examples and the output has to be copied inside the docstring the corresponding function.

We demonstrate this way of proceeding with the following simple example. We have slimmed down the previous module, so that only the function fib is left:

```python
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

We now call this module in an interactive Python shell and do some calculations:

```python
from fibonacci import fib
fib(0)
```

Output: 0

```python
fib(1)
```

Output: 1

```python
fib(10)
```

Output: 55

```python
fib(15)
```

Output: 610

We copy the complete session of the interactive shell into the docstring of our function. To start the module doctest we have to call the method testmod(), but only if the module is called standalone. The complete module looks like this now:

```python
import doctest

def fib(n):
```

```python
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

if __name__ == "__main__":
    doctest.testmod()
```

If we start our module directly like this

```python
import fibonacci_doctest
```

we get no output, because everything is okay.

To see how doctest works, if something is wrong, we place an error in our code: We change again

a, b = 0, 1

into

a, b = 1, 1

and save the file as fibonacci_doctest1.py

```python
exec(open("fibonacci_doctest1.py").read())
```

```
    ****************************************************************
    File "__main__", line 7, in __main__.fib
    Failed example:
        fib(0)
    Expected:
        0
    Got:
        1
    ****************************************************************
    File "__main__", line 11, in __main__.fib
    Failed example:
        fib(10)
    Expected:
        55
    Got:
        89
    ****************************************************************
    File "__main__", line 13, in __main__.fib
    Failed example:
        fib(15)
    Expected:
        610
    Got:
        987
    ****************************************************************
    1 items had failures:
       3 of   4 in __main__.fib
    ***Test Failed*** 3 failures.
```

The output depicts all the calls, which return faulty results. We can see the call with the arguments in the line following "Failed example:". We can see the expected value for the argument in the line following "Expected:". The output shows us the newly calculated value as well. We can find this value behind "Got:"

TEST-DRIVEN DEVELOPMENT (TDD)

In the previous chapters, we tested functions, which we had already been finished. What about testing code you haven't yet written? You think that this is not possible? It is not only possible, it is the underlying idea of test-driven development. In the extreme case, you define tests before you start coding the actual source code. The program developer writes an automated test case which defines the desired "behaviour" of a function. This test case will - that's the idea behind the approach - initially fail, because the code has still to be written.

The major problem or difficulty of this approach is the task of writing suitable tests. Naturally, the perfect test would check all possible inputs and validate the output. Of course, this is generally not always feasible.

We have set the return value of the fib function to 0 in the following example:

```python
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """

    return 0

if __name__ == "__main__":
    doctest.testmod()
```

```
********************************************************************
File "__main__", line 9, in __main__.fib
Failed example:
    fib(1)
Expected:
    1
Got:
    0
********************************************************************
File "__main__", line 11, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    0
********************************************************************
File "__main__", line 13, in __main__.fib
Failed example:
    fib(15)
Expected:
    610
Got:
    0
********************************************************************
1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```

It hardly needs mentioning that the function returns only wrong return values except for fib(0).

Now we have to keep on writing and changing the code for the function fib until it passes the test.

This test approach is a method of software development, which is called test-driven development.

UNITTEST

The Python module unittest is a unit testing framework, which is based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework. The module contains the core framework classes that form the basis of the test cases and suites (TestCase, TestSuite and so on), and also a text-based utility class for running the tests and reporting the results (TextTestRunner). The most obvious difference to the module "doctest" lies in the fact that the test cases of the module "unittest" are not defined inside the module, which has to be tested. The major advantage is clear: program documentation and test descriptions are separate from each other. The price you have to pay on the other hand, is an increase of work to create the test cases.

We will use our module fibonacci once more to create a test case with unittest. To this purpose we create a file fibonacci_unittest.py. In this file we have to import unittest and the module which has to be tested, i.e. fibonacci.

Furthermore, we have to create a class with an arbitrary name - we will call it "FibonacciTest" - which inherits from unittest.TestCase. The test cases are defined in this class by using methods. The name of these methods is arbitrary, but has to start with test. In our method "testCalculation" we use the method assertEqual from the class TestCase. assertEqual(first, second, msg = None) checks, if expression "first" is equal to the expression "second". If the two expressions are not equal, msg will be output, if msg is not None.

```python
import unittest
from fibonacci1 import fib

class FibonacciTest(unittest.TestCase):


    def testCalculation(self):
        self.assertEqual(fib(0), 0)
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(5), 5)
        self.assertEqual(fib(10), 55)
        self.assertEqual(fib(20), 6765)


if name == "main":
     unittest.main()
```

If we call this test case, we get the following output:

```
$ python3 fibonacci_unittest.py
.
-------------------------------------------------------------------
---
Ran 1 test in 0.000s

OK
```

This is usually the desired result, but we are now interested what happens in the error case. Therefore we will create our previous error again. We change again the well-known line:

```python
a, b = 0, 1
```

will be changed to

```
    a, b = 1, 1


    import unittest
    from fibonacci3 import fib

    class FibonacciTest(unittest.TestCase):

        def testCalculation(self):
            self.assertEqual(fib(0), 0)
            self.assertEqual(fib(1), 1)
            self.assertEqual(fib(5), 5)
            self.assertEqual(fib(10), 55)
            self.assertEqual(fib(20), 6765)

    if __name__ == "__main__":
        unittest.main()
```

Now the test result looks like this:

```
    $ python3 fibonacci_unittest.py
    F
    ======================================================================
    ===
    FAIL: testCalculation (__main__.FibonacciTest)
    ----------------------------------------------------------------------
    ---
    Traceback (most recent call last):
      File "fibonacci_unittest.py", line 7, in testCalculation
        self.assertEqual(fib(0), 0)
    AssertionError: 1 != 0


    ----------------------------------------------------------------------
    ---
    Ran 1 test in 0.000s

    FAILED (failures=1)
```

The first statement in testCalculation has created an exception. The other assertEqual calls had not been executed. We correct our error and create a new one. Now all the values will be correct, except if the input argument is 20:

```
    def fib(n):
        """ Iterative Fibonacci Function """
```

```
        a, b = 0, 1
        for i in range(n):
            a, b = b, a + b
        if n == 20:
            a = 42
        return a
```

The output of a test run looks now like this:

```
$ python3 fibonacci_unittest.py
blabal
```

F

```
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 12, in testCalculation
    self.assertEqual(fib(20), 6765)
AssertionError: 42 != 6765
```

```
Ran 1 test in 0.000s

FAILED (failures=1)
```

All the statements of testCalculation have been executed, but we haven't seen any output, because everything was okay:

```
self.assertEqual(fib(0), 0)
self.assertEqual(fib(1), 1)
self.assertEqual(fib(5), 5)
```

Methods of the Class TestCase We now have a closer look at the class TestCase.

Method	Meaning
setUp()	Hook method for setting up the test fixture before exercising it. This method is called before calling the implemented test methods.
tearDown()	Hook method for deconstructing the class fixture after running all tests in the class.
assertEqual(self, first, second, msg=None)	The test fails if the two objects are not equal as determined by the '==' operator.
assertAlmostEqual(self, first, second, places=None, msg=None, delta=None)	The test fails if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit). If the two objects compare equal then they will automatically compare almost equal.
assertCountEqual(self, first, second, msg=None)	An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

Method	Meaning
	self.assertEqual(Counter(list(first)), Counter(list(second))) Example: [0, 1, 1] and [1, 0, 1] compare equal, because the number of ones and zeroes are the same. [0, 0, 1] and [0, 1] compare unequal, because zero appears twice in the first list and only once in the second list.
assertDictEqual(self, d1, d2, msg=None)	Both arguments are taken as dictionaries and they are checked if they are equal.
assertTrue(self, expr, msg=None)	Checks if the expression "expr" is True.
assertGreater(self, a, b, msg=None)	Checks, if a > b is True.
assertGreaterEqual(self, a, b, msg=None)	Checks if a ≥ b
assertFalse(self, expr, msg=None)	Checks if expression "expr" is False.
assertLess(self, a, b, msg=None)	Checks if a < b
assertLessEqual(self, a, b, msg=None)	Checks if a ≤ b
assertIn(self, member, container, msg=None)	Checks if a in b
assertIs(self, expr1, expr2, msg=None)	Checks if "a" is "b"
assertIsInstance(self, obj, cls, msg=None)	Checks if isinstance(obj, cls).
assertIsNone(self, obj, msg=None)	Checks if "obj is None"
assertIsNot(self, expr1, expr2, msg=None)	Checks if "a" is not "b"

Method	Meaning
`assertIsNotNone(self, obj, msg=None)`	Checks if obj is not equal to None
`assertListEqual(self, list1, list2, msg=None)`	Lists are checked for equality.
`assertMultiLineEqual(self, first, second, msg=None)`	Assert that two multi-line strings are equal.
`assertNotRegexpMatches(self, text, unexpected_regexp, msg=None)`	Fails, if the text Text "text" of the regular expression unexpected_regexp matches.
`assertTupleEqual(self, tuple1, tuple2, msg=None)`	Analogous to assertListEqual

We expand our previous example by a setUp and a tearDown method:

```
import unittest
from fibonacci1 import fib

class FibonacciTest(unittest.TestCase):

    def setUp(self):
        self.fib_elems = ( (0,0), (1,1), (2,1), (3,2), (4,3),
(5,5) )
        print ("setUp executed!")

    def testCalculation(self):
        for (i,val) in self.fib_elems:
            self.assertEqual(fib(i), val)

    def tearDown(self):
        self.fib_elems = None
        print ("tearDown executed!")

if __name__ == "__main__":
    unittest.main()
```

A call returns the following results:

```
$ python3 fibonacci_unittest2.py
setUp executed!
tearDown executed!
.
----------------------------------------------------------------
---
Ran 1 test in 0.000s
```

Most of the TestCase methods have an optional parameter "msg". It's possible to return an additional description of an error with "msg".

Exercises

1. Exercise:

Can you find a problem in the following code?

```python
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively

    fib(0)
    0
    fib(1)
    1
    fib(10)
    55
    fib(40)
    102334155


    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    doctest.testmod()
```

Answer:

The doctest is okay. The problem is the implementation of the fibonacci function. This recursive approach is "highly" inefficient. You need a lot of patience to wait for the termination of the test. The number of hours, days or weeks depending on your computer. ☺

Footnotes:

*The aphorism in full length: "Errare (Errasse) humanum est, sed in errare (errore) perseverare diabolicum." (To err is human, but to persist in it is diabolic")

** In computer science, the syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Writing "if" as "iff" is an example for syntax error, both in programming and in the English language.

INTRODUCTION
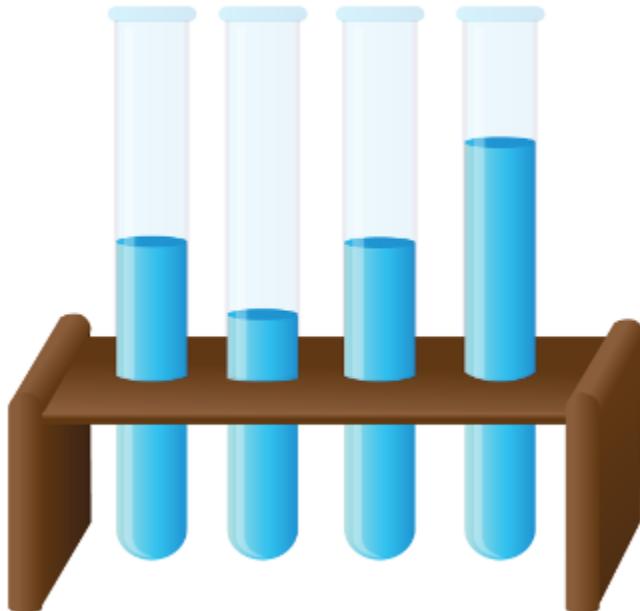
pytest can be used for all types and levels of software testing. Many projects – amongst them Mozilla and Dropbox - switched from unittest or nose to pytest.

A SIMPLE FIRST EXAMPLE WITH PYTEST

Test files which pytest will use for testing have to start with test_ or end with _test.py We will demonstrate the way of working by writing a test file test_fibonacci.py for a file fibonacci.py. Both files are in one directory:

The first file is the file which should be tested. We assume that it is saved as fibonacci_p.py:

```
def fib(n):
    old, new = 0, 1
    for _ in range(n):
        old, new = new, old + new
    return old
```

Now, we have to provide the code for the file test_fibonacci.py. This file will be used by 'pytest':

```
from fibonacci_p import fib


def test_fib():
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(10) == 55
```

We call pytest in a command shell in the directory where the two file shown above reside:

```
pytest
```

The result of this code can be seen in the following:

```
============================= test session starts
==============================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0
rootdir: /home/bernd/, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 1 ite
m

test_fibonacci.py
.                                                              [100%]

========================== 1 passed in 0.01 seconds
==========================
```

We create now an erroneous version of fib. We change the two start values from 0 and 1 to the values 2 and 1. This is the beginning of the Lucas sequence, but as we want to implement the Fibonacci sequence this is wrong. This way, we can study how pytest behaves in this case:

```
def fib(n):
    old, new = 2, 1
    for _ in range(n):
        old, new = new, old + new
    return old
```

Calling 'pytest' with this erroneous implementation of fibonacci gives us the following results:

```
$ pytest  ============================= test session starts
==============================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0
rootdir: /home/bernd/, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 1 ite
m

test_fibonacci.py
F                                                              [100%]

=================================== FAILURES
===================================
_____ test_fib
```

```
        _____

            def test_fib():
        >           assert fib(0) == 0
        E           assert 2 == 0
        E            +  where 2 = fib(0)

        test_fibonacci.py:5: AssertionError
        =========================== 1 failed in 0.03 seconds
        ===========================
```

ANOTHER PYTEST EXAMPLE

We will get closer to 'reality' in our next example. In a real life scenario, we will usually have more than one file and for each file we may have a corresponding test file. Each test file may contain various tests. We have various files in our example folder ex2:

The files to be tested:

- fibonacci.py
- foobar_plus.py
- foobar.py

The test files:

- test_fibonacci.py
- test_foobar_plus.py
- test_foobar.py

We start 'pytest' in the directory 'ex2' and get the following results:

```
$ pytest
==================== test session starts =====================
platform linux -- Python 3.7.3, pytest-4.3.1, py-1.8.0, plugg
y-0.9.0
rootdir: /home/bernd/ex2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.3.0, arra
ydiff-0.3
collected 4 item
s

test_fibonacci.py
.                                                      [ 25%]
test_foobar.py
..                                                     [ 75%]
```

```
test_foobar_plus.py
.                                                            [100%]


==================== 4 passed in 0.05 seconds =================
```

ANOTHER PYTEST EXAMPLE

It is possible to execute only tests, which contain a given substring in their name. The substring is determined by a Python expression This can be achieved with the call option „-k"

```
pytest -k
```

The call

```
pytest -k foobar
```

will only execute the test files having the substring 'foobar' in their name. In this case, they are test_foobar.py and test_foobar_plus.py:

```
$ pytest -k foobar
============================ test session starts
============================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0
rootdir: /home/bernd/ex2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 3 items / 1 deselecte
d

test_foobar.py
.                                                            [ 50%]
test_foobar_plus.py
.                                                            [100%]

==================== 2 passed, 1 deselected in 0.01 seconds
====================
```

We will select now only the files containing 'plus' and 'fibo'

```
 pytest -k 'plus or fibo'
```

```
$ pytest -k 'plus or fibo'
============================ test session starts
```

```
==============================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0
rootdir: /home/bernd/ex2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 3 items / 1 deselecte
d

test_fibonacci.py
.                                                        [ 50%]
test_foobar_plus.py
.                                                        [100%]

==================== 2 passed, 1 deselected in 0.01 seconds
====================
```

MARKERS IN PYTEST

Test functions can be marked or tagged by decorating them with 'pytest.mark.'.

Such a marker can be used to select or deselect test functions. You can see the markers which exist for your test suite by typing

```
 $ pytest --markers
```

```
$ pytest --markers
@pytest.mark.openfiles_ignore: Indicate that open files should be i
gnored for this test

@pytest.mark.remote_data: Apply to tests that require data from rem
ote servers

@pytest.mark.internet_off: Apply to tests that should only run whe
n network access is deactivated

@pytest.mark.filterwarnings(warning): add a warning filter to the g
iven test. see https://docs.pytest.org/en/latest/warnings.html#pyte
st-mark-filterwarnings

@pytest.mark.skip(reason=None): skip the given test function with a
n optional reason. Example: skip(reason="no way of currently testin
g this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eva
l(condition) results in a True value.  Evaluation happens within th
```

e module global context. Example: skipif('sys.platform == "win3
2"') skips the test if we are on the win32 platform. see https://do
cs.pytest.org/en/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, s
trict=False): mark the test function as an expected failure if eva
l(condition) has a True value. Optionally specify a reason for bett
er reporting and run=False if you don't even want to execute the te
st function. If only specific exception(s) are expected, you can li
st them in raises, and if the test fails in other ways, it will be
reported as a true failure. See https://docs.pytest.org/en/latest/s
kipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test functio
n multiple times passing in different arguments in turn. argvalues
generally needs to be a list of values if argnames specifies only o
ne name or a list of tuples of values if argnames specifies multipl
e names. Example: @parametrize('arg1', [1,2]) would lead to two cal
ls of the decorated test function, one with arg1=1 and another wit
h arg1=2.see https://docs.pytest.org/en/latest/parametrize.html fo
r more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tes
ts as needing all of the specified fixtures. see https://docs.pytes
t.org/en/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such tha
t the plugin machinery will try to call it first/as early as possib
le.

@pytest.mark.trylast: mark a hook implementation function such tha
t the plugin machinery will try to call it last/as late as possibl
e.

This contains also custom defined markers!

REGISTERING MARKERS

Since pytest version 4.5 markers have to be registered.They can be registered in the init file pytest.ini, placed in the test directory. We register the markers 'slow' and 'crazy', which we will use in the following example:

[pytest] markers = slow: mark a test as a 'slow' (slowly) running test crazy: stupid function to test :-)

We add a recursive and inefficient version rfib to our fibonacci module and mark the corresponding test routine with slow, besides this rfib is marked with crazy as well:

```python
# content of fibonacci.py

def fib(n):
    old, new = 0, 1
    for i in range(n):
        old, new = new, old + new
    return old


def rfib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return rfib(n-1) + rfib(n-2)
```

The corresponding test file:

```python
""" content of test_fibonacci.py """

import pytest
from fibonacci import fib, rfib

def test_fib():
    assert fib(0) == 0
    assert fib(1) == 1
    assert fib(34) == 5702887

@pytest.mark.crazy
@pytest.mark.slow
def test_rfib():
    assert fib(0) == 0
    assert fib(1) == 1
    assert rfib(34) == 5702887
```

Besides this we will add the files foobar.py and test_foobar.py as well. We mark the test functions in test_foobar.py as crazy.

```python
# content of foobar.py

def foo():
    return "foo"
```

```
def bar():
    return "bar"
```

This is the correponding test file:

content of test_fooba r.py

```
import pytest
from foobar import foo, bar

@pytest.mark.crazy
def test_foo():
    assert foo() == "foo"

@pytest.mark.crazy
def test_bar():
    assert bar() == "bar"

</pre
```

We will start tests now depending on the markers. Let's start all tests, which are not marked as slow:

```
$ pytest -svv -k "slow"
==================================== test session starts
======================================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0 -- /home/bernd/python
cachedir: .pytest_cache
rootdir: /home/bernd/ex_tagging, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 4 items / 3 deselecte
d

test_fibonacci.py::test_rfib PASSED

=========================== 1 passed, 3 deselected in 7.05 second
s ============================
```

We will run now only the tests which are not marked as slow or crazy:

```
$ pytest -svv -k "not slow and not crazy"
======================= test session starts =======================
platform linux -- Python 3.7.1, pytest-4.0.2, py-1.7.0, plugg
y-0.8.0 -- /home/bernd/
cachedir: .pytest_cache
rootdir: /home/bernd//ex_tagging, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arra
ydiff-0.3
collected 4 items / 3 deselecte
d

test_fibonacci.py::test_fib PASSED

===================== 1 passed, 3 deselected in 0.01 seconds
===================
```

SKIPIF MARKER

If you wish to skip a function conditionally then you can use skipif. In the following example the function test_foo is marked with a skipif. The function will not be executed, if the Python version is 3.6.x:

```python
import pytest
import sys
from foobar import foo, bar


@pytest.mark.skipif(
    sys.version_info[0] == 3 and sys.version_info[1] == 6,
    reason="Python version has to be higher than 3.5!")
def test_foo():
    assert foo() == "foo"

@pytest.mark.crazy
def test_bar():
    assert bar() == "bar"
```

Instead of a conditional skip we can also use an unconditional skip. This way, we can always skip. We can add a reason. The following example shows how this can be accomplished by marking the function test_bar with a skip marker. The reason we give is that it is "even fooer than foo":

```python
import pytest
import sys
```

```
from foobar import foo, bar

@pytest.mark.skipif(
    sys.version_info[0] == 3 and sys.version_info[1] == 6,
    reason="Python version has to be higher than 3.5!")
def test_foo():
    assert foo() == "foo"


@pytest.mark.skip(reason="Even fooer than foo, so we skip!")
def test_bar():
    assert bar() == "bar"
```

If we call pytest on this code, we get the following output:

```
$ pytest -v
================ test session starts ===============
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0 -- /home/bernd/python
cachedir: .pytest_cache
rootdir: /home/bernd/ex_tagging2, inifile: pytest.ini
collected 4 item
s


test_fibonacci.py::test_fib PASSE
D
         [ 25%]
test_fibonacci.py::test_rfib PASSE
D
        [ 50%]
test_foobar.py::test_foo SKIPPE
D
           [ 75%]
test_foobar.py::test_bar PASSE
D
              [100%]

============= 3 passed, 1 skipped in 0.01 seconds =============
```

PARAMETRIZATION WITH MARKERS

We will demonstrate parametrization with markers with our Fibonacci function.

```
# content of fibonacci.py

def fib(n):
    old, new = 0, 1
    for _ in range(n):
        old, new = new, old + new
    return old
```

We write a pytest test function which will test against this fibonacci function with various values:

```
# content of the file test_fibonacci.py

import pytest

from fibonacci import fib

@pytest.mark.parametrize(
    'n, res', [(0, 0),
               (1, 1),
               (2, 1),
               (3, 2),
               (4, 3),
               (5, 5),
               (6, 8)])
def test_fib(n, res):
    assert fib(n) == res
```

When we call pytest, we get the following results:

```
$ pytest -v
=========================== test session starts
============================
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0 -- /home/bernd/python
cachedir: .pytest_cache
rootdir: /home/bernd/ex_parametrization1
collected 7 item
s


test_fibonacci.py::test_fib[0-0] PASSE
D                                                              [ 14%]
test_fibonacci.py::test_fib[1-1] PASSE
D                                                              [ 28%]
```

```
test_fibonacci.py::test_fib[2-1] PASSE
D                                                               [ 42%]
test_fibonacci.py::test_fib[3-2] PASSE
D                                                               [ 57%]
test_fibonacci.py::test_fib[4-3] PASSE
D                                                               [ 71%]
test_fibonacci.py::test_fib[5-5] PASSE
D                                                               [ 85%]
test_fibonacci.py::test_fib[6-8] PASSE
D                                                               [100%]

========================= 7 passed in 0.01 seconds
=========================
```

The numbers inside the square brackets on front of the word "PASSED" are the values of 'n' and 'res'.

PRINTS IN FUNCTIONS

If there are prints in the functions which we test, we will not see this output in our pytests, unless we call pytest with the option "-s". To demonstrate this we will add a print line to our fibonacci function:

```
def fib(n):
    old, new = 0, 1
    for _ in range(n):
        old, new = new, old + new
    print("result: ", old)
    return old
```

Calling "pytest -s -v" will deliver the following output:

```
=============== test session starts ===============
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0 -- /home/bernd/python
cachedir: .pytest_cache
rootdir: /home/bernd/ex_parametrization1
collected 7 item
s


test_fibonacci.py::test_fib[0-0] result:  0
PASSED
test_fibonacci.py::test_fib[1-1] result:  1
PASSED
test_fibonacci.py::test_fib[2-1] result:  1
```

```
PASSED
test_fibonacci.py::test_fib[3-2] result:  2
PASSED
test_fibonacci.py::test_fib[4-3] result:  3
PASSED
test_fibonacci.py::test_fib[5-5] result:  5
PASSED
test_fibonacci.py::test_fib[6-8] result:  8
PASSED

============= 7 passed in 0.01 seconds =================
```

COMMAND LINE OPTIONS / FIXTURES

We will write a test version for our fibonacci function which depends on command line arguments. We can add custom command line options to pytest with the pytest_addoption hook that allows us to manage the command line parser and the setup for each test. At first, we have to write a file conftest.py with the functions cmdopt and pytest_addoption:

```python
import pytest

def pytest_addoption(parser):
    parser.addoption("--cmdopt",
                     action="store",
                     default="full",
                     help="'num' of tests or full")

@pytest.fixture
def cmdopt(request):
    return request.config.getoption("--cmdopt")
```

The code for our fibonacci test module looks like. The test_fif function has a parameter 'cmdopt' which gets the parameter option:

```python
from fibonacci import fib

results = [0, 1, 1, 2, 3, 5, 8, 13, 21,
          34, 55, 89, 144, 233, 377]


def test_fib(cmdopt):
    if cmdopt == "full":
        num = len(results)
```

```
        else:
            num = len(results)
            if int(cmdopt) < len(results):
                num = int(cmdopt)
        for i in range(num):
            assert fib(i) == results[i]
```

We can call it now with various options, as we can see in the following:

```
$ pytest -q --cmdopt=full -v -s
============ test session starts ===============
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0
rootdir: /home/bernd/Dropbox (Bodenseo)/kurse/python_en/examples/py
test/ex_cmd_line
collected 1 ite
m


test_fibonacci.py running 15 tests!
.

============ 1 passed in 0.01 seconds ============



$ pytest -q --cmdopt=6 -v -s
============ test session starts ==============
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0
rootdir: /home/bernd/Dropbox (Bodenseo)/kurse/python_en/examples/py
test/ex_cmd_line
collected 1 ite
m


test_fibonacci.py running  6 tests!
.

========================== 1 passed in 0.01 seconds
==============================
```
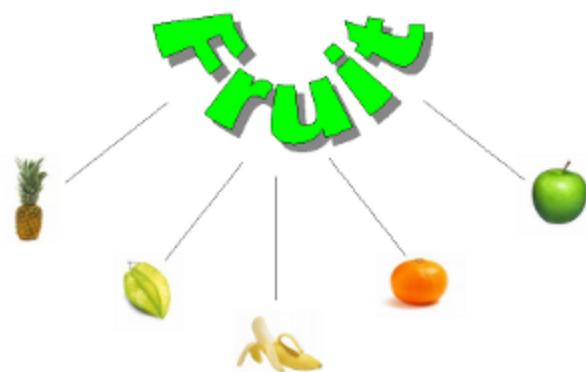
Let's put an error in our test results: results = [0, 1, 1, 2, 3, 1001, 8,…] Calling pytest with 'pytest -q --cmdopt=10 -v -s' gives us the following output:

```
================== test session starts ==================
platform linux -- Python 3.6.9, pytest-5.0.1, py-1.8.0, pluggy-0.1
2.0 rootdir: /home/bernd/ex_cmd_line collected 1 item test_fibonacc
i.py running 10 tests! F =============== FAILURES
==================== _____ test_fib _____ cm
dopt = '10' def test_fib(cmdopt): if cmdopt == "full": num = len(re
sults) else: num = len(results) if int(cmdopt) < len(results): num
= int(cmdopt) print(f"running {num:2d} tests!") for i in range(nu
m): > assert fib(i) == results[i] E assert 5 == 1001 E + where 5 =
fib(5) test_fibonacci.py:16: AssertionError =============== 1 fail
ed in 0.03 seconds =================
```

OBJECT-ORIENTED PROGRAMMING

Though Python is an object-oriented language without fuss or quibble, we have so far intentionally avoided the treatment of object-oriented programming (OOP) in the previous chapters of our Python tutorial. We skipped OOP, because we are convinced that it is easier and more fun to start learning Python without having to know about all the details of object-oriented programming.

Even though we have avoided OOP, it has nevertheless always been present in the exercises and examples of our course. We used objects and methods from classes without properly explaining their OOP background. In this chapter, we will catch up on what has been missing so far. We will provide an introduction into the principles of object oriented programming in general and into the specifics of the OOP approach of Python. OOP is one of the most powerful tools of Python, but nevertheless you don't have to use it, i.e. you can write powerful and efficient programs without it as well.

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

You will learn to know the four major principles of object-orientation and the way Python deals with them in the next section of this tutorial on object-oriented programming:

- Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

Before we start the section about the way OOP is used in Python, we want to give you a general idea about object-oriented programming. For this purpose, we would like to draw your attention to a public library. Let's think about a huge one, like the "British Library" in London or the "New York Public Library" in New York. If it helps, you can imagine the libraries in Paris, Berlin, Ottawa or Toronto* as well. Each of these contain an organized collection of books, periodicals, newspapers, audiobooks, films and so on.

Generally, there are two opposed ways of keeping the stock in a library. You can use a "closed access" method that is the stock is not displayed on open shelves. In this system, trained staff brings the books and other publications to the users on demand. Another way of running a library is open-access shelving, also known as "open shelves". "Open" means open to all users of the library not only specially trained staff. In this case the books are openly displayed. Imperative languages like C could be seen as open-access shelving libraries. The user can do everything. It's up to the user to find the books and to put them back at the right shelf. Even though this is great for the user, it might lead to serious problems in the long run. For example some books will be misplaced, so it's hard to find them again. As you may have guessed already, "closed access" can be compared to object oriented programming. The analogy can be seen like this: The books and other publications, which a library offers, are like the data in an object-oriented program. Access to the books is restricted like access to the data is restricted in OOP. Getting or returning a book is only possible via the staff. The staff functions like the methods in OOP, which control the access to the data. So, the data, - often called attributes, - in such a program can be seen as being hidden and protected by a shell, and it can only be accessed by special functions, usually called methods in the OOP context. Putting the data behind a "shell" is called *Encapsulation*. So a library can be regarded as a class and a book is an instance or an object of this class. Generally speaking, an object is defined by a class. A class is a formal description of how an object is designed, i.e. which attributes and methods it has. These objects are called instances as well. The expressions are in most cases used synonymously. A class should not be confused with an object.

OOP IN PYTHON

Even though we haven't talked about classes and object orientation in previous chapters, we have worked with classes all the time. In fact, everything is a class in Python. Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." (Blog, The History of Python, February 27, 2009) In other words, "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

```python
x = 42
type(x)
```

Output: `int`

```python
y = 4.34
type(y)
```

Output: `float`

```python
def f(x):
    return x + 1
type(f)
```

Output: `function`

```python
import math
type(math)
```

Output: `module`

One of the many integrated classes in Python is the list class, which we have quite often used in our exercises and examples. The list class provides a wealth of methods to build lists, to access and change elements, or to remove elements:

```python
x = [3,6,9]
y = [45, "abc"]
print(x[1])
```

```
6
```
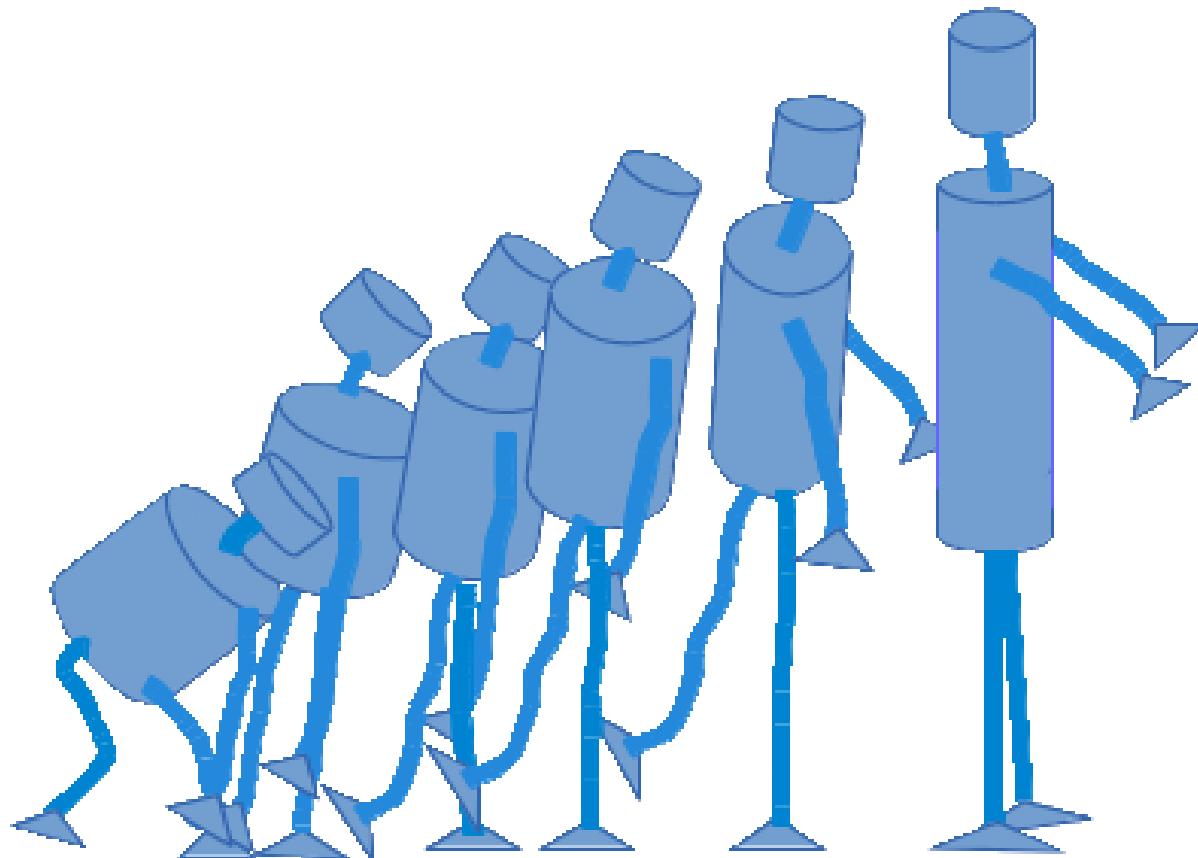
```
x[1] = 99
x.append(42)
last = y.pop()
print(last)
```

```
abc
```

The variables x and y of the previous example denote two instances of the list class. In simplified terms, we have said so far that "x and y are lists". We will use the terms "object" and "instance" synonymously in the following chapters, as it is often done**.

pop and append of the previous example are methods of the list class. pop returns the top (or you might think of it as the "rightest") element of the list and removes this element from the list. We will not explain how Python has implemented lists internally. We don't need this information, because the list class provides us with all the necessary methods to access the data indirectly. That is, the encapsulation details are encapsulated. We will learn about encapsulation later.

A MINIMAL CLASS IN PYTHON

We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

```
class Robot:
    pass
```

We can realize the fundamental syntactical structure of a class in Python: A class consists of two parts: the header and the body. The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "Robot" in our case. The class name is followed by a listing of other class names, which are classes from which the defined class inherits. These classes are called superclasses, base classes or sometimes parent classes. If you look at our example, you will see that this listing of superclasses is not obligatory. You don't have to bother about inheritance and superclasses for now. We will introduce them later.

The body of a class consists of an indented block of statements. In our case a single statement, the "pass" statement.

A class object is created, when the definition is left normally, i.e. via the end. This is basically a wrapper around the contents of the namespace created by the class definition.

It's hard to believe, especially for C++ or Java programmers, but we have already defined a complete class with just three words and two lines of code. We are capable of using this class as well:

```python
class Robot:
    pass

if __name__ == "__main__":
    x = Robot()
    y = Robot()
    y2 = y
    print(y == y2)
    print(y == x)
```

```
True
False
```

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y.

ATTRIBUTES

Those who have learned already another object-oriented language, must have realized that the terms attributes and properties are usually used synonymously. It may even be used in the definition of an attribute, like Wikipedia does: "In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such."

Even in normal English usage the words "attribute" and "property" can be used in some cases as synonyms. Both can have the meaning "An attribute, feature, quality, or characteristic of something or someone". Usually an "attribute" is used to denote a specific ability or characteristic which something or someone has, like black hair, no hair, or a quick perception, or "her quickness to grasp new tasks". So, think a while about your outstanding attributes. What about your "outstanding properties"? Great, if one of your strong points is your ability to quickly understand and adapt to new situations! Otherwise, you would not be learning Python!

Let's get back to Python: We will learn later that properties and attributes are essentially different things in Python. This subsection of our tutorial is about attributes in Python. So far our robots have no attributes. Not even a name, like it is customary for ordinary robots, isn't it? So, let's implement a name attribute. "type designation", "build year" etc. are easily conceivable as further attributes as well***.

Attributes are created inside a class definition, as we will soon learn. We can dynamically create arbitrary new attributes for existing instances of a class. We do this by joining an arbitrary name to the instance name, separated by a dot ".". In the following example, we demonstrate this by creating an attribute for the name and the year built:

```python
class Robot:
    pass

x = Robot()
y = Robot()

x.name = "Marvin"
x.build_year = "1979"

y.name = "Caliban"
y.build_year = "1993"

print(x.name)
```

    Marvin

```python
print(y.build_year)
```

    1993

As we have said before: This is not the way to properly create instance attributes. We introduced this example, because we think that it may help to make the following explanations easier to understand.

If you want to know, what's happening internally: The instances possess dictionaries `__dict__`, which they use to store their attributes and their corresponding values:

```python
x.__dict__
```

Output: `{'name': 'Marvin', 'build_year': '1979'}`

```
y.__dict__
```
Output: `{'name': 'Caliban', 'build_year': '1993'}`

Attributes can be bound to class names as well. In this case, each instance will possess this name as well. Watch out, what happens, if you assign the same name to an instance:

```python
class Robot(object):
    pass
```

```python
x = Robot()
Robot.brand = "Kuka"
x.brand
```
Output: `'Kuka'`

```python
x.brand = "Thales"
Robot.brand
```
Output: `'Kuka'`

```python
y = Robot()
y.brand
```
Output: `'Kuka'`

```python
Robot.brand = "Thales"
```

```python
y.brand
```
Output: `'Thales'`

```python
x.brand
```
Output: `'Thales'`

If you look at the `__dict__` dictionaries, you can see what's happening.

```python
x.__dict__
```
Output: `{'brand': 'Thales'}`

```python
y.__dict__
```
Output: `{}`

```python
Robot.__dict__
```

```
mappingproxy({'__module__': '__main__',
              '__dict__': <attribute '__dict__' of 'Robot' ob
jects>,
              '__weakref__': <attribute '__weakref__' of 'Rob
ot' objects>,
              '__doc__': None,
              'brand': 'Thales'})
```

If you try to access y.brand, Python checks first, if "brand" is a key of the y. `__dict__` dictionary. If it is not, Python checks, if "brand" is a key of the Robot. `__dict__`. If so, the value can be retrieved.

If an attribute name is not in included in either of the dictionary, the attribute name is not defined. If you try to access a non-existing attribute, you will raise an AttributeError:

```
x.energy
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-82fa0f11497d> in <module>
----> 1 x.energy

AttributeError: 'Robot' object has no attribute 'energy'
```

By using the function getattr, you can prevent this exception, if you provide a default value as the third argument:

```
getattr(x, 'energy', 100)
```

100

Binding attributes to objects is a general concept in Python. Even function names can be attributed. You can bind an attribute to a function name in the same way, we have done so far to other instances of classes:

```
def f(x):
    return 42

f.x = 42
print(f.x)
```

```
42
```

This can be used as a replacement for the static function variables of C and C++, which are not possible in Python. We use a counter attribute in the following example:

```python
def f(x):
    f.counter = getattr(f, "counter", 0) + 1
    return "Monty Python"


for i in range(10):
    f(i)

print(f.counter)
```

```
10
```

Some uncertainty may arise at this point. It is possible to assign attributes to most class instances, but this has nothing to do with defining classes. We will see soon how to assign attributes when we define a class.

To properly create instances of classes, we also need methods. You will learn in the following subsection of our Python tutorial, how you can define methods.

METHODS

Methods in Python are essentially functions in accordance with Guido's saying "first-class everything".

Let's define a function "hi", which takes an object "obj" as an argument and assumes that this object has an attribute "name". We will also define our basic Robot class again:

```python
def hi(obj):
    print("Hi, I am " + obj.name + "!")

class Robot:
    pass


x = Robot()
x.name = "Marvin"
hi(x)
```
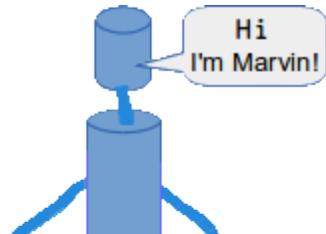
```
Hi, I am Marvin!
```

We will now bind the function „hi" to a class attribute „say_hi"!

```python
def hi(obj):
        print("Hi, I am " + obj.name)
```

```
class Robot:
    say_hi = hi


x = Robot()
x.name = "Marvin"
Robot.say_hi(x)
```

```
Hi, I am Marvin
```

"say_hi" is called a method. Usually, it will be called like this:

```
    x.say_hi()
```

It is possible to define methods like this, but you shouldn't do it.

The proper way to do it:

- Instead of defining a function outside of a class definition and binding it to a class attribute, we define a method directly inside (indented) of a class definition.
- A method is "just" a function which is defined inside a class.
- The first parameter is used a reference to the calling instance.
- This parameter is usually called self.
- Self corresponds to the Robot object x.

We have seen that a method differs from a function only in two aspects:

- It belongs to a class, and it is defined within a class
- The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "self".

As a matter of fact, "self" is not a Python keyword. It's just a naming convention! So C++ or Java programmers are free to call it "this", but this way they are risking that others might have greater difficulties in understanding their code!

Most other object-oriented programming languages pass the reference to the object (self) as a hidden parameter to the methods.

You saw before that the calls Robot.say_hi(x)". and "x.say_hi()" are equivalent. "x.say_hi()" can be seen as an "abbreviated" form, i.e. Python automatically binds it to the instance name. Besides this "x.say_hi()" is the usual way to call methods in Python and in other object oriented languages.

For a Class C, an instance x of C and a method m of C the following three method calls are equivalent:

- type(x).m(x, ...)
- C.m(x, ...)
- x.m(...)

Before you proceed with the following text, you may mull over the previous example for awhile. Can you figure out, what is wrong in the design?

There is more than one thing about this code, which may disturb you, but the essential problem at the moment is the fact that we create a robot and that after the creation, we shouldn't forget about naming it! If we forget it, say_hi will raise an error.

We need a mechanism to initialize an instance right after its creation. This is the `__init__`-method, which we cover in the next section.

THE __INIT__ METHOD

We want to define the attributes of an instance right after its creation. `__init__` is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to chose another name. `__init__` is one of the so-called magic methods, we will get to know it with some more details later. The `__init__` method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The `__init__` method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

```python
class A:
    def __init__(self):
        print("__init__ has been executed!")

x = A()
```

```
__init__ has been executed!
```

We add an `__init__`-method to our robot class:

```python
class Robot:

    def __init__(self, name=None):
        self.name = name

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")


x = Robot()
x.say_hi()
y = Robot("Marvin")
```

```
y.say_hi()
```

```
Hi, I am a robot without a name
Hi, I am Marvin
```

DATA ABSTRACTION, DATA ENCAPSULATION, AND INFORMATION HIDING

DEFINITIONS OF TERMS

Data Abstraction, Data Encapsulation and Information Hiding are often synonymously used in books and tutorials on OOP. However, there is a difference. Encapsulation is seen as the bundling of data with the methods that operate on that data. Information hiding on the other hand is the principle that some internal information or data is "hidden", so that it can't be accidentally changed. Data encapsulation via methods doesn't necessarily mean that the data is hidden. You might be capable of accessing and seeing the data anyway, but using the methods is recommended. Finally, data abstraction is present, if both data hiding and data encapsulation is used. In other words, data abstraction is the broader term:

Data Abstraction = Data Encapsulation + Data Hiding

Encapsulation is often accomplished by providing two kinds of methods for attributes: The methods for retrieving or accessing the values of attributes are called getter methods. Getter methods do not change the values of attributes, they just return the values. The methods used for changing the values of attributes are called setter methods.
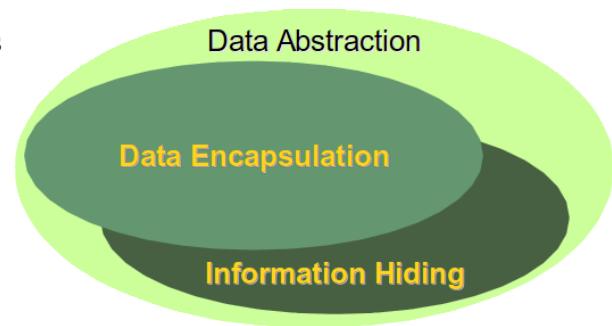
We will define now a Robot class with a Getter and a Setter for the name attribute. We will call them get_name and set_name accordingly.

```python
class Robot:

    def __init__(self, name=None):
        self.name = name

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.name = name
```

```
    def get_name(self):
        return self.name


x = Robot()
x.set_name("Henry")
x.say_hi()
y = Robot()
y.set_name(x.get_name())
print(y.get_name())
```

```
Hi, I am Henry
Henry
```

Before you go on, you can do a little exercise. You can add an additional attribute "build_year" with Getters and Setters to the Robot class.

```
class Robot:

    def __init__(self,
                 name=None,
                 build_year=None):
        self.name = name
        self.build_year = build_year

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")
        if self.build_year:
            print("I was built in " + str(self.build_year))
        else:
            print("It's not known, when I was created!")

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_build_year(self, by):
        self.build_year = by
```

```
    def get_build_year(self):
        return self.build_year


x = Robot("Henry", 2008)
y = Robot()
y.set_name("Marvin")
x.say_hi()
y.say_hi()
```

```
Hi, I am Henry
I was built in 2008
Hi, I am Marvin
It's not known, when I was created!
```

There is still something wrong with our Robot class. The Zen of Python says: "There should be one-- and preferably only one --obvious way to do it." Our Robot class provides us with two ways to access or to change the "name" or the "build_year" attribute. This can be prevented by using private attributes, which we will explain later.

__STR__- AND __REPR__-METHODS

We will have a short break in our treatise on data abstraction for a quick side-trip. We want to introduce two important magic methods "__str__" and "__repr__", which we will need in future examples. In the course of this tutorial, we have already encountered the __str__ method. We had seen that we can depict various data as string by using the str function, which uses "magically" the internal __str__ method of the corresponding data type. __repr__ is similar. It also produces a string representation.

```
l = ["Python", "Java", "C++", "Perl"]
print(l)
```

```
['Python', 'Java', 'C++', 'Perl']
```

```
str(l)
```

Output: "['Python', 'Java', 'C++', 'Perl']"

```
repr(l)
```

Output: "['Python', 'Java', 'C++', 'Perl']"

```
d = {"a":3497, "b":8011, "c":8300}
print(d)
```

```
{'a': 3497, 'b': 8011, 'c': 8300}
```

```
str(d)
```

Output: `"{'a': 3497, 'b': 8011, 'c': 8300}"`

```
repr(d)
```

Output: `"{'a': 3497, 'b': 8011, 'c': 8300}"`

```
x = 587.78
str(x)
```

Output: `'587.78'`

```
repr(x)
```

Output: `'587.78'`

If you apply str or repr to an object, Python is looking for a corresponding method `__str__` or `__repr__` in the class definition of the object. If the method does exist, it will be called. In the following example, we define a class A, having neither a `__str__` nor a `__repr__` method. We want to see, what happens, if we use print directly on an instance of this class, or if we apply str or repr to this instance:

```
class A:
    pass
```

```
a = A()
print(a)
```

```
<__main__.A object at 0x0000016B162A2688>
```

```
print(repr(a))
```

```
<__main__.A object at 0x0000016B162A2688>
```

```
print(str(a))
```

```
<__main__.A object at 0x0000016B162A2688>
```

```
a
```

Output: `<__main__.A at 0x16b162a2688>`

As both methods are not available, Python uses the default output for our object "a".

If a class has a `__str__` method, the method will be used for an instance x of that class, if either the function str is applied to it or if it is used in a print function. `__str__` will not be used, if repr is called, or if we try to output the value directly in an interactive Python shell:

```
class A:
```

```python
    def __str__(self):
        return "42"

a = A()

print(repr(a))

<__main__.A object at 0x0000016B162A4108>

print(str(a))

42

a
```

```
<__main__.A at 0x16b162a4108>
```

Otherwise, if a class has only the `__repr__` method and no `__str__` method, `__repr__` will be applied in the situations, where `__str__` would be applied, if it were available:

```python
class A:
    def __repr__(self):
        return "42"

a = A()
print(repr(a))
print(str(a))
a
```

```
42
42
```
42

A frequently asked question is when to use `__repr__` and when `__str__`. `__str__` is always the right choice, if the output should be for the end user or in other words, if it should be nicely printed. `__repr__` on the other hand is used for the internal representation of an object. The output of `__repr__` should be - if feasible - a string which can be parsed by the python interpreter. The result of this parsing is in an equal object. That is, the following should be true for an object "o":

```python
o == eval(repr(o))
```

This is shown in the following interactive Python session:

```python
l = [3,8,9]
```

```
s = repr(l)
s
```

Output: `'[3, 8, 9]'`

```
l == eval(s)
```

Output: True

```
l == eval(str(l))
```

Output: True

We show in the following example with the datetime module that eval can only be applied on the strings created by repr:

```
import datetime
today = datetime.datetime.now()
str_s = str(today)
eval(str_s)
```

```
Traceback (most recent call last):

  File "C:\Users\melis\Anaconda3\lib\site-packages\IPython\core\in
teractiveshell.py", line 3326, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-57-52b036e82a64>", line 4, in <module>
    eval(str_s)

  File "<string>", line 1
    2020-06-11 17:48:26.391635
            ^
SyntaxError: invalid token
```

```
repr_s = repr(today)
t = eval(repr_s)
type(t)
```

Output: datetime.datetime

We can see that eval(repr_s) returns again a datetime.datetime object. The string created by str can't be turned into a datetime.datetime object by parsing it.

We will extend our robot class with a repr method. We dropped the other methods to keep this example simple:

```python
class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot('" + self.name + "', " +  str(self.build_yea
r) +  ")"

if __name__ == "__main__":
    x = Robot("Marvin", 1979)

    x_str = str(x)
    print(x_str)
    print("Type of x_str: ", type(x_str))
    new = eval(x_str)
    print(new)
    print("Type of new:", type(new))
```

```
Robot('Marvin', 1979)
Type of x_str:  <class 'str'>
Robot('Marvin', 1979)
Type of new: <class '__main__.Robot'>
```

x_str has the value Robot('Marvin', 1979). eval(x_str) converts it again into a Robot instance.

Now it's time to extend our class with a user friendly `__str__` method:

```python
class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot('" + self.name + "', " +  str(self.build_yea
r) +  ")"

    def __str__(self):
        return "Name: " + self.name + ", Build Year: " +  str(sel
f.build_year)

if __name__ == "__main__":
```

```
    x = Robot("Marvin", 1979)

    x_str = str(x)
    print(x_str)
    print("Type of x_str: ", type(x_str))
    new = eval(x_str)
    print(new)
    print("Type of new:", type(new))
```

Name: Marvin, Build Year: 1979
Type of x_str: <class 'str'>

Traceback (most recent call last):

  File "C:\Users\melis\Anaconda3\lib\site-packages\IPython\core\in
teractiveshell.py", line 3326, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-60-c8082321922a>", line 19, in <module>
    new = eval(x_str)

  File "<string>", line 1
    Name: Marvin, Build Year: 1979
        ^
SyntaxError: invalid syntax

When we start this program, we can see that it is not possible to convert our string x_str, created via str(x),
into a Robot object anymore.

We show in the following program that x_repr can still be turned into a Robot object:

```
class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot(\"" + self.name + "\"," +  str(self.build_ye
ar) +  ")"

    def __str__(self):
        return "Name: " + self.name + ", Build Year: " +  str(sel
f.build_year)
```

```python
if __name__ == "__main__":
    x = Robot("Marvin", 1979)

    x_repr = repr(x)
    print(x_repr, type(x_repr))
    new = eval(x_repr)
    print(new)
    print("Type of new:", type(new))
```

```
Robot("Marvin",1979) <class 'str'>
Name: Marvin, Build Year: 1979
Type of new: <class '__main__.Robot'>
```

PUBLIC, - PROTECTED-, AND PRIVATE ATTRIBUTES

Who doesn't know those trigger-happy farmers from films. Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but than he receives a $50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area**.

Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar on the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.

We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under any circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!

Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.

We have the same classification again in object-oriented programming:

- Private attributes should only be used by the owner, i.e. inside of the class definition itself.
- Protected (restricted) Attributes may be used, but at your own risk. Essentially, they should only be used under certain conditions.
- Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course. There are two ways to restrict the access to class attributes:

- First, we can prefix an attribute name with a leading underscore "_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, they write a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
- Second, we can prefix an attribute name with two leading underscores "__". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside the class definition itself*.

To summarize the attribute types:

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside a class definition.
_name	Protected	Protected attributes should not be used outside the class definition, unless inside a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself.

We want to demonstrate the behaviour of these attribute types with an example class:

```python
class A():

    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
```

We store this class (attribute_tests.py) and test its behaviour in the following interactive Python shell:

```python
from attribute_tests import A
x = A()
x.pub
```

Output: `'I am public'`

```python
x.pub = x.pub + " and my value can be changed"
x.pub
```

`'I am public and my value can be changed'`

```
x._prot
```

`'I am protected'`

```
x.__priv
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-f75b36b98afa> in <module>
----> 1 x.__priv

AttributeError: 'A' object has no attribute '__priv'
```

The error message is very interesting. One might have expected a message like " `__priv` is private". We get the message "AttributeError: 'A' object has no attribute `__priv` instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task is rewriting our Robot class. Though we have Getter and Setter methods for the name and the build_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of self.name and self.build_year by self.`__name` and `self.__build_year`.

The listing of our revised class:

```python
class Robot:

    def __init__(self, name=None, build_year=2000):
        self.__name = name
        self.__build_year = build_year

    def say_hi(self):
        if self.__name:
            print("Hi, I am " + self.__name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.__name = name

    def get_name(self):
```

```python
        return self.__name

    def set_build_year(self, by):
        self.__build_year = by

    def get_build_year(self):
        return self.__build_year

    def __repr__(self):
        return "Robot('" + self.__name + "', " +  str(self.__buil
d_year) +  ")"

    def __str__(self):
        return "Name: " + self.__name + ", Build Year: " +  str(se
lf.__build_year)


if __name__ == "__main__":
    x = Robot("Marvin", 1979)
    y = Robot("Caliban", 1943)
    for rob in [x, y]:
        rob.say_hi()
        if rob.get_name() == "Caliban":
            rob.set_build_year(1993)
        print("I was built in the year " + str(rob.get_build_yea
r()) + "!")
```

```
Hi, I am Marvin
I was built in the year 1979!
Hi, I am Caliban
I was built in the year 1993!
```

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

This may look like the following class:

```python
    class A():

        def __init__(self, x, y):
            self.__x = x
            self.__y = y

        def GetX(self):
            return self.__x
```

```
    def GetY(self):
        return self.__y

    def SetX(self, x):
        self.__x = x

    def SetY(self, y):
        self.__y = y
```

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic code this way, as you will learn soon.

DESTRUCTOR

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method `__del__`. It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with `__init__` and `__del__`:

```python
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print ("Robot has been destroyed")


if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

```
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
Robot has been destroyed
Robot has been destroyed
```

The usage of the `__del__` method is very problematic. If we change the previous code to personalize the deletion of a robot, we create an error:

```python
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print (self.name + " says bye-bye!")


if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

```
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
```

```
Exception ignored in: <function Robot.__del__ at 0x0000024A13CDC8B
8>
Traceback (most recent call last):
  File "<ipython-input-18-7dedd1b7f17b>", line 7, in __del__
AttributeError: 'Robot' object has no attribute 'name'
Exception ignored in: <function Robot.__del__ at 0x0000024A13CDC8B
8>
Traceback (most recent call last):
  File "<ipython-input-18-7dedd1b7f17b>", line 7, in __del__
AttributeError: 'Robot' object has no attribute 'name'
```

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

Footnotes: + The picture on the right side is taken in the Library of the Court of Appeal for Ontario, located downtown Toronto in historic Osgoode Hall

++ "Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value." (excerpt from the official Python Language Reference)

+++ "attribute" stems from the Latin verb "attribuere" which means "to associate with"

++++ http://www.wickedlocal.com/x937072506/tJogger-ticketed-for-trespassing-in-the-Winchester-Fells-kicks-back

+++++ There is a way to access a private attribute directly. In our example, we can do it like this: x._Robot__build_year You shouldn't do this under any circumstances!

CLASS AND INSTANCE ATTRIBUTES

CLASS ATTRIBUTES

Instance attributes are owned by the specific instances of a class. That is, for two different instances, the instance attributes are usually different. You should by now be familiar with this concept which we introduced in our previous chapter.

We can also define attributes at the class level. Class attributes are attributes which are owned by the class itself. They will be shared by all the instances of the class. Therefore they have the same value for every instance. We define class attributes outside all the methods, usually they are placed at the top, right below the class header.

In the following interactive Python session, we can see that the class attribute "a" is the same for all instances, in our examples "x" and "y". Besides this, we see that we can access a class attribute via an instance or via the class name:

```python
class A:
    a = "I am a class attribute!"

x = A()
y = A()
x.a
```
Output: `'I am a class attribute!'`

```python
y.a
```
Output: `'I am a class attribute!'`

```python
A.a
```
Output: `'I am a class attribute!'`

But be careful, if you want to change a class attribute, you have to do it with the notation ClassName.AttributeName. Otherwise, you will create a new instance variable. We demonstrate this in the following example:

```python
class A:
```

```
    a = "I am a class attribute!"

x = A()
y = A()
x.a = "This creates a new instance attribute for x!"
```

```
y.a
```
Output: `'I am a class attribute!'`

```
A.a
```
Output: `'I am a class attribute!'`

```
A.a = "This is changing the class attribute 'a'!"
```

```
A.a
```
Output: `"This is changing the class attribute 'a'!"`

```
y.a
```
Output: `"This is changing the class attribute 'a'!"`

```
x.a
# but x.a is still the previously created instance variable
```
Output: `'This creates a new instance attribute for x!'`

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

```
x.__dict__
```
Output: `{'a': 'This creates a new instance attribute for x!'}`

```
 y.__dict__
```
Output: `{}`

```
A.__dict__
```

`mappingproxy({'__module__': '__main__',`
                    `'a': "This is changing the class attribute`
      `'a'!",`
                    `'__dict__': <attribute '__dict__' of 'A' object`
      `s>,`
                    `'__weakref__': <attribute '__weakref__' of 'A'`
      `objects>,`
                    `'__doc__': None})`

```
x.__class__.__dict__
```
`mappingproxy({'__module__': '__main__',`
                    `'a': "This is changing the class attribute`
      `'a'!",`
                    `'__dict__': <attribute '__dict__' of 'A' object`
      `s>,`
                    `'__weakref__': <attribute '__weakref__' of 'A'`
      `objects>,`
                    `'__doc__': None})`

EXAMPLE WITH CLASS ATTRIBUTES

Isaac Asimov devised and introduced the so-called "Three Laws of Robotics" in 1942. The appeared in his story "Runaround". His three laws have been picked up by many science fiction writers. As we have started manufacturing robots in Python, it's high time to make sure that they obey Asimovs three laws. As they are the same for every instance, i.e. robot, we will create a class attribute Three_Laws. This attribute is a tuple with the three laws.

```python
class Robot:

    Three_Laws = (
"""A robot may not injure a human being or, through inaction, allo
w a human being to come to harm.""",
"""A robot must obey the orders given to it by human beings, excep
t where such orders would conflict with the First Law.,""",
"""A robot must protect its own existence as long as such protecti
on does not conflict with the First or Second Law."""
)
    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    # other methods as usual
```

As we mentioned before, we can access a class attribute via instance or via the class name. You can see in the

following that we don't need an instance:

```python
from robot_asimov import Robot
for number, text in enumerate(Robot.Three_Laws):
    print(str(number+1) + ":\n" + text)
```

1:
A robot may not injure a human being or, through inaction, allow
a human being to come to harm.
2:
A robot must obey the orders given to it by human beings, except w
here such orders would conflict with the First Law.,
3:
A robot must protect its own existence as long as such protection
does not conflict with the First or Second Law.

In the following example, we demonstrate, how you can count instance with class attributes. All we have to do
is

- to create a class attribute, which we call "counter" in our example
- to increment this attribute by 1 every time a new instance is created
- to decrement the attribute by 1 every time an instance is destroyed

```python
class C:

    counter = 0

    def __init__(self):
        type(self).counter += 1

    def __del__(self):
        type(self).counter -= 1

if __name__ == "__main__":
    x = C()
    print("Number of instances: : " + str(C.counter))
    y = C()
    print("Number of instances: : " + str(C.counter))
    del x
    print("Number of instances: : " + str(C.counter))
    del y
    print("Number of instances: : " + str(C.counter))
```

```
Number of instances: : 1
Number of instances: : 2
Number of instances: : 1
Number of instances: : 0
```

Principially, we could have written C.counter instead of type(self).counter, because type(self) will be evaluated to "C" anyway. However we will understand later, that type(self) makes sense, if we use such a class as a superclass.

STATIC METHODS

We used class attributes as public attributes in the previous section. Of course, we can make public attributes private as well. We can do this by adding the double underscore again. If we do so, we need a possibility to access and change these private class attributes. We could use instance methods for this purpose:

```python
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def RobotInstances(self):
        return Robot.__counter


if __name__ == "__main__":
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
```

```
1
2
```

This is not a good idea for two reasons: First of all, because the number of robots has nothing to do with a single robot instance and secondly because we can't inquire the number of robots before we create an instance. If we try to invoke the method with the class name Robot.RobotInstances(), we get an error message, because it needs an instance as an argument:

```
Robot.RobotInstances()
```

```
-------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-f53600e3296e> in <module>
----> 1 Robot.RobotInstances()

TypeError: RobotInstances() missing 1 required positional argument: 'self'
```

The next idea, which still doesn't solve our problem, is omitting the parameter "self":

```python
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def RobotInstances():
        return Robot.__counter
```

Now it's possible to access the method via the class name, but we can't call it via an instance:

```python
from static_methods2 import Robot
Robot.RobotInstances()
```

Output: 0

```python
x = Robot()
x.RobotInstances()
```

```
-------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-4d5e11c3474a> in <module>
      1 x = Robot()
----> 2 x.RobotInstances()

TypeError: RobotInstances() takes 0 positional arguments but 1 was given
```

The call "x.RobotInstances()" is treated as an instance method call and an instance method needs a reference to the instance as the first parameter.

So, what do we want? We want a method, which we can call via the class name or via the instance name without the necessity of passing a reference to an instance to it.

The solution lies in static methods, which don't need a reference to an instance. It's easy to turn a method into a static method. All we have to do is to add a line with "@staticmethod" directly in front of the method header. It's the decorator syntax.

You can see in the following example that we can now use our method RobotInstances the way we want:

```python
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def RobotInstances():
        return Robot.__counter


if __name__ == "__main__":
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())
```

```
0
1
2
2
```

CLASS METHODS

Static methods shouldn't be confused with class methods. Like static methods class methods are not bound to instances, but unlike static methods class methods are bound to a class. The first parameter of a class method is a reference to a class, i.e. a class object. They can be called via an instance or the class name.

```python
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @classmethod
    def RobotInstances(cls):
        return cls, Robot.__counter


if __name__ == "__main__":
```

```
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())
```

```
(<class '__main__.Robot'>, 0)
(<class '__main__.Robot'>, 1)
(<class '__main__.Robot'>, 2)
(<class '__main__.Robot'>, 2)
```

The use cases of class methods:

- They are used in the definition of the so-called factory methods, which we will not cover here.
- They are often used, where we have static methods, which have to call other static methods. To do this, we would have to hard code the class name, if we had to use static methods. This is a problem, if we are in a use case, where we have inherited classes.

The following program contains a fraction class, which is still not complete. If you work with fractions, you need to be capable of reducing fractions, e.g. the fraction 8/24 can be reduced to 1/3. We can reduce a fraction to lowest terms by dividing both the numerator and denominator by the Greatest Common Divisor (GCD).

We have defined a static gcd function to calculate the greatest common divisor of two numbers. the greatest common divisor (gcd) of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the 'GCD of 8 and 24 is 8. The static method "gcd" is called by our class method "reduce" with "cls.gcd(n1, n2)". "CLS" is a reference to "fraction".

```
class fraction(object):

    def __init__(self, n, d):
        self.numerator, self.denominator = fraction.reduce(n, d)


    @staticmethod
    def gcd(a,b):
        while b != 0:
            a, b = b, a%b
        return a

    @classmethod
    def reduce(cls, n1, n2):
        g = cls.gcd(n1, n2)
        return (n1 // g, n2 // g)

    def __str__(self):
```

```
        return str(self.numerator)+'/'+str(self.denominator)
```

Using this class:

```
from fraction1 import fraction
x = fraction(8,24)
print(x)
```

```
1/3
```

CLASS METHODS VS. STATIC METHODS AND INSTANCE METHODS

Our last example will demonstrate the usefulness of class methods in inheritance. We define a class `Pet` with a method `about`. This method should give some general class information. The class `Cat` will be inherited both in the subclass `Dog` and `Cat`. The method `about` will be inherited as well. We will demonstrate that we will encounter problems, if we define the method `about` as a normal instance method or as a static method. We will start by defining `about` as an instance method:

```
class Pet:
    _class_info = "pet animals"

    def about(self):
        print("This class is about " + self._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```

This may look alright at first at first glance. On second thought we recognize the awful design. We had to create instances of the `Pet`, `Dog` and `Cat` classes to be able to ask what the class is about. It would be a

lot better, if we could just write `Pet.about()`, `Dog.about()` and `Cat.about()` to get the previous result. We cannot do this. We will have to write `Pet.about(p)`, `Dog.about(d)` and `Cat.about(c)` instead.

Now, we will define the method `about` as a "staticmethod" to show the disadvantage of this approach. As we have learned previously in our tutorial, a staticmethod does not have a first parameter with a reference to an object. So about will have no parameters at all. Due to this, we are now capable of calling "about" without the necessity of passing an instance as a parameter, i.e. `Pet.about()`, `Dog.about()` and `Cat.about()`. Yet, a problem lurks in the definition of `about`. The only way to access the class info `_class_info` is putting a class name in front. We arbitrarily put in `Pet`. We could have put there `Cat` or `Dog` as well. No matter what we do, the solution will not be what we want:

```python
class Pet:
    _class_info = "pet animals"

    @staticmethod
    def about():
        print("This class is about " + Pet._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

Pet.about()
Dog.about()
Cat.about()
```

```
This class is about pet animals!
This class is about pet animals!
This class is about pet animals!
```

In other words, we have no way of differenciating between the class `Pet` and its subclasses `Dog` and `Cat`. The problem is that the method `about` does not know that it has been called via the `Pet` the `Dog` or the `Cat` class.

A classmethod is the solution to all our problems. We will decorate `about` with a classmethod decorator instead of a staticmethod decorator:

```python
class Pet:
    _class_info = "pet animals"

    @classmethod
```

```
    def about(cls):
        print("This class is about " + cls._class_info + "!")



class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

Pet.about()
Dog.about()
Cat.about()
```

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```

PROPERTIES VS. GETTERS AND SETTERS

PROPERTIES

Getters(also known as 'accessors') and setters (aka. 'mutators') are used in many object oriented programming languages to ensure the principle of data encapsulation. Data encapsulation - as we have learnt in our introduction on Object Oriented Programming of our tutorial - is seen as the bundling of data with the methods that operate on them. These methods are of course the *getter* for retrieving the data and the *setter* for changing the data. According to this principle, the attributes of a class are made private to hide and protect them.

Unfortunately, it is widespread belief that a proper Python class should encapsulate private attributes by using getters and setters. As soon as one of these programmers introduces a new attribute, he or she will make it a private variable and creates "automatically" a getter and a setter for this attribute. Such programmers may even use an editor or an IDE, which automatically creates getters and setters for all private attributes. These tools even warn the programmer if she or he uses a public attribute! Java programmers will wrinkle their brows, screw up their noses, or even scream with horror when they read the following: The Pythonic way to introduce attributes is to make them public.

We will explain this later. First, we demonstrate in the following example, how we can design a class in a Javaesque way with getters and setters to encapsulate the private attribute `self.__x`:

```python
class P:

    def __init__(self, x):
        self.__x = x

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x
```

We can see in the following demo session how to work with this class and the methods:

```
from mutators import P
p1 = P(42)
p2 = P(4711)
p1.get_x()
```
Output: 42

```
p1.set_x(47)
p1.set_x(p1.get_x()+p2.get_x())
p1.get_x()
```
Output: 4758

What do you think about the expression "p1.set_x(p1.get_x()+p2.get_x())"? It's ugly, isn't it? It's a lot easier to write an expression like the following, if we had a public attribute x:

```
p1.x = p1.x + p2.x
```

Such an assignment is easier to write and above all easier to read than the Javaesque expression.

Let's rewrite the class P in a Pythonic way. No getter, no setter and instead of the private attribute `self.__x` we use a public one:

```
class P:

    def __init__(self,x):
        self.x = x
```

Beautiful, isn't it? Just three lines of code, if we don't count the blank line!

```
from p import P
p1 = P(42)
p2 = P(4711)
p1.x
```
Output: 42

```
p1.x = 47
p1.x = p1.x + p2.x
p1.x
```
Output: 4758

"But, but, but, but, but ... ", we can hear them howling and screaming, "But there is NO data ENCAPSULATION!" Yes, in this case there is no data encapsulation. We don't need it in this case. The only thing get_x and set_x in our starting example did was "getting the data through" without doing anything

additionally.

But what happens if we want to change the implementation in the future? This is a serious argument. Let's assume we want to change the implementation like this: The attribute x can have values between 0 and 1000. If a value larger than 1000 is assigned, x should be set to 1000. Correspondingly, x should be set to 0, if the value is less than 0.

It is easy to change our first P class to cover this problem. We change the set_x method accordingly:

```python
class P:

    def __init__(self, x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

The following Python session shows that it works the way we want it to work:

```python
from mutators1 import P
p1 = P(1001)
p1.get_x()
```
Output: 1000

```python
p2 = P(15)
p2.get_x()
```
Output: 15

```python
p3 = P(-1)
p3.get_x()
```
Output: 0

But there is a catch: Let's assume we designed our class with the public attribute and no methods:

```
class P2:

    def __init__(self, x):
        self.x = x
```

People have already used it a lot and they have written code like this:

```
p1 = P2(42)
p1.x = 1001
p1.x
```

Output: 1001

If we would change P2 now in the way of the class P, our new class would break the interface, because the attribute x will not be available anymore. That's why in Java e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.

But Python offers a solution to this problem. The solution is called *properties*!

The class with a property looks like this:

```
class P:

    def __init__(self, x):
        self.x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

A method which is used for getting a value is decorated with "@property", i.e. we put this line directly in front of the header. The method which has to function as the setter is decorated with "@x.setter". If the function had been called "f", we would have to decorate it with "@f.setter". Two things are noteworthy: We just put the code line "self.x = x" in the __init__ method and the property method x is used to check the limits of the

values. The second interesting thing is that we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". We have learned in a previous chapter of our course that this is not possible. It works here due to the decorating:

```python
from p2 import P
p1 = P(1001)
p1.x
```

Output: `1000`

```python
p1.x = -12
p1.x
```

Output: `0`

Alternatively, we could have used a different syntax without decorators to define the property. As you can see, the code is definitely less elegant and we have to make sure that we use the getter function in the `__init__` method again:

```python
class P:

    def __init__(self, x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(get_x, set_x)
```

There is still another problem in the most recent version. We have now two ways to access or change the value of x: Either by using "p1.x = 42" or by "p1.set_x(42)". This way we are violating one of the fundamentals of Python: "There should be one-- and preferably only one --obvious way to do it." (see Zen of Python)

We can easily fix this problem by turning the getter and the setter methods into private methods, which can't be accessed anymore by the users of our class P:

```python
class P:
```

```python
    def __init__(self, x):
        self.__set_x(x)

    def __get_x(self):
        return self.__x

    def __set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(__get_x, __set_x)
```
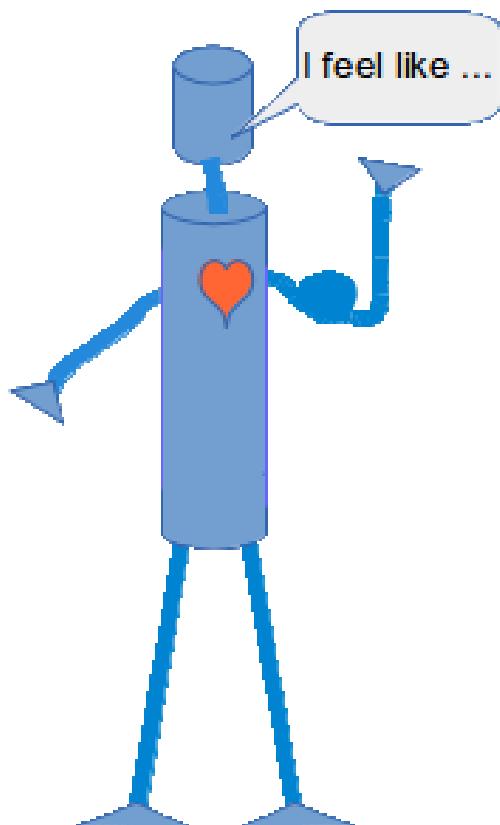
Even though we fixed this problem by using a private getter and setter, the version with the decorator "@property" is the Pythonic way to do it!

From what we have written so far, and what can be seen in other books and tutorials as well, we could easily get the impression that there is a one-to-one connection between properties (or mutator methods) and the attributes, i.e. that each attribute has or should have its own property (or getter-setter-pair) and the other way around. Even in other object oriented languages than Python, it's usually not a good idea to implement a class like that. The main reason is that many attributes are only internally needed and creating interfaces for the user of the class increases unnecessarily the usability of the class. The possible user of a class shouldn't be "drowned" with umpteen - of mainly unnecessary - methods or properties!

I feel like ...

The following example shows a class, which has internal attributes, which can't be accessed from outside. These are the private attributes `self.__potential_physical` and `self.__potential_psychic`. Furthermore we show that a property can be deduced from the values of more than one attribute. The property "condition" of our example returns the condition of the robot in a descriptive string. The condition depends on the sum of the values of the psychic and the physical conditions of the robot.

```python
class Robot:
```

```python
    def __init__(self, name, build_year, lk = 0.5, lp = 0.5 ):
        self.name = name
        self.build_year = build_year
        self.__potential_physical = lk
        self.__potential_psychic = lp

    @property
    def condition(self):
        s = self.__potential_physical + self.__potential_psychic
        if s <= -1:
            return "I feel miserable!"
        elif s <= 0:
            return "I feel bad!"
        elif s <= 0.5:
            return "Could be worse!"
        elif s <= 1:
            return "Seems to be okay!"
        else:
            return "Great!"

if __name__ == "__main__":
    x = Robot("Marvin", 1979, 0.2, 0.4 )
    y = Robot("Caliban", 1993, -0.4, 0.3)
    print(x.condition)
    print(y.condition)
```

```
Seems to be okay!
I feel bad!
```

PUBLIC INSTEAD OF PRIVATE ATTRIBUTES

Let's summarize the usage of private and public attributes, getters and setters, and properties: Let's assume that we are designing a new class and we pondering about an instance or class attribute "OurAtt", which we need for the design of our class. We have to observe the following issues:

- Will the value of "OurAtt" be needed by the possible users of our class?
- If not, we can or should make it a private attribute.
- If it has to be accessed, we make it accessible as a public attribute
- We will define it as a private attribute with the corresponding property, if and only if we have to do some checks or transformation of the data. (As an example, you can have a look again at our class P, where the attribute has to be in the interval between 0 and 1000, which is ensured by the property "x")
- Alternatively, you could use a getter and a setter, but using a property is the Pythonic way to deal with it!

Let's assume we defined "OurAtt" as a public attribute. Our class has been successfully used by other users for

quite a while.

```python
class OurClass:

    def __init__(self, a):
        self.OurAtt = a


x = OurClass(10)
print(x.OurAtt)
```

```
10
```

Now comes the point which frightens some traditional OOPistas out of their wits: Imagine "OurAtt" has been used as an integer. Now, our class has to ensure that "OurAtt" has to be a value between 0 and 1000? Without property, this is really a horrible scenario! Due to properties it's easy: We create a property version of "OurAtt".

```python
class OurClass:

    def __init__(self, a):
        self.OurAtt = a

    @property
    def OurAtt(self):
        return self.__OurAtt

    @OurAtt.setter
    def OurAtt(self, val):
        if val < 0:
            self.__OurAtt = 0
        elif val > 1000:
            self.__OurAtt = 1000
        else:
            self.__OurAtt = val


x = OurClass(10)
print(x.OurAtt)
```

```
10
```

This is great, isn't it? You can start with the simplest implementation imaginable, and you are free to later migrate to a property version without having to change the interface! So properties are not just a replacement

for getters and setters!

Something else you might have already noticed: For the users of a class, properties are syntactically identical to ordinary attributes.

In the previous chapter of our tutorial, we learned how to create and use properties in a class. The main objective was to understand them as a way to get rid of explicit getters and setters and have a simple class interface. This is usually enough to know for most programmers and for practical use cases and they will not need more.

If you want to know more about how 'property' works, you can go one step further with us. By doing this, you can improve your coding skills and get a deeper insight and understanding of Python. We will have a look at the way the "property" decorator could be implemented in Python code. (It is implemented in C code in reality!) By doing this, the way of working will be clearer. Everything is based on the descriptor protocol, which we will explain later.

We define a class with the name 'our_property' so that it will not be mistaken for the existing 'property' class. This class can be used like the 'real' property class.

```python
class our_property:
    """ emulation of the property class
        for educational purposes """

    def __init__(self,
                 fget=None,
                 fset=None,
                 fdel=None,
                 doc=None):
        """Attributes of 'our_decorator'
        fget
            function to be used for getting
            an attribute value
        fset
            function to be used for setting
            an attribute value
        fdel
            function to be used for deleting
            an attribute
        doc
            the docstring
        """
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
```

```python
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

We need another class to use the previously defined class and to demonstrate how the property class decorator works. To continue the tradition of the previous chapters of our Python tutorial we will again write a Robot class. We will define a property in this example class to demonstrate the usage of our previously defined property class or better 'our_decorator' class. When you run the code, you can see `__init__` of 'our_property' will be called 'fget' set to a reference to the 'getter' function of 'city'. The attribute 'city' is an instance of the 'our_property' class. The 'our'property' class provides another decorator 'setter' for the setter functionality. We apply this with '@city.setter'

```python
class Robot:

    def __init__(self, city):
        self.city = city
```

```python
    @our_property
    def city(self):
        print("The Property 'city' will be returned now:")
        return self.__city

    @city.setter
    def city(self, city):
        print("'city' will be set")
        self.__city = city
```

'Robot.city' is an instance of the 'our_property' class as we can see in the following:

```python
type(Robot.city)
```

Output: __main__.our_property

If you change the line '@our_property' to '@property' the program will behave totally the same, but it will be using the original Python class 'property' and not our 'our_property' class. We will create instances of the Robot class in the following Python code:

```python
print("Instantiating a Root and setting 'city' to 'Berlin'")
robo = Robot("Berlin")
print("The value is: ", robo.city)
print("Our robot moves now to Frankfurt:")
robo.city = "Frankfurt"
print("The value is: ", robo.city)
```

```
Instantiating a Root and setting 'city' to 'Berlin'
'city' will be set
The Property 'city' will be returned now:
The value is:  Berlin
Our robot moves now to Frankfurt:
'city' will be set
The Property 'city' will be returned now:
The value is:  Frankfurt
```

Let's make our property implementation a little bit more talkative with some print functions to see what is going on. We also change the name to 'chatty_property' for obvious reasons:

```python
class chatty_property:
    """ emulation of the property class
        for educational purposes """
```

```python
    def __init__(self,
                 fget=None,
                 fset=None,
                 fdel=None,
                 doc=None):

        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        print("\n__init__ called with:)")
        print(f"fget={fget}, fset={fset}, fdel={fdel}, doc={doc}")
        if doc is None and fget is not None:
            print(f"doc set to docstring of {fget.__name__} metho
d")
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__do
c__)

    def setter(self, fset):
        print(type(self))
        return type(self)(self.fget, fset, self.fdel, self.__do
c__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__do
```

```
c__)
```

```python
class Robot:

    def __init__(self, city):
        self.city = city

    @chatty_property
    def city(self):
        """ city attribute of Robot """
        print("The Property 'city' will be returned now:")
        return self.__city

    @city.setter
    def city(self, city):
        print("'city' will be set")
        self.__city = city
```

```
__init__ called with:)
fget=<function Robot.city at 0x000002825C9420D8>, fset=None, fde
l=None, doc=None
doc set to docstring of city method
<class '__main__.chatty_property'>

__init__ called with:)
fget=<function Robot.city at 0x000002825C9420D8>, fset=<function R
obot.city at 0x000002825C9425E8>, fdel=None, doc= city attribute o
f Robot
```

```python
robo = Robot("Berlin")
```

```
'city' will be set
```

In the previous two chapters of our Python tutorial, we learned how to use Python properties and even how to implement a custom-made property class. In this chapter you will learn the details of descriptors.

Descriptors were introduced to Python in version 2.2. That time the "What's New in Python2.2" mentioned: "The one big idea underlying the new class model is that an API for describing the attributes of an object using descriptors has been formalized. Descriptors specify the value of an attribute, stating whether it's a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic constructs."
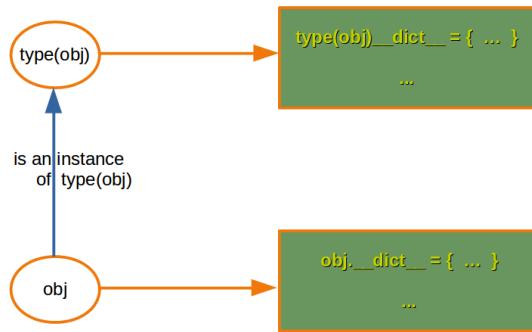
A descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`.

If any of those methods are defined for an object, it is said to be a descriptor.
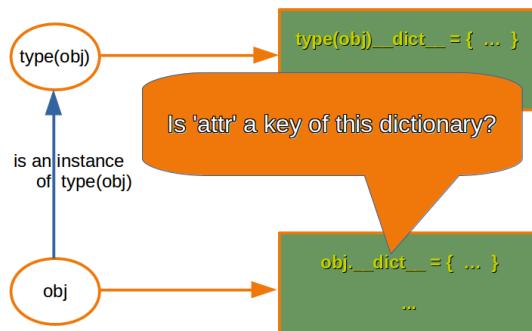
Their purpose consists in providing programmers with the ability to add managed attributes to classes. The descriptors are introduced to get, set or delete attributes from the object's `__dict__` dictionary via the above mentioned methods. Accessing a class attribute will start the lookup chain.

Let a closer look at what is happening. Assuming we have an object `obj` : What happens if we try to access an attribute (or property) `ap` ? "Accesssing" the attribute means to "get" the value, so the attribute is used for example in a print function or inside of an expression. Both the `obj` and the class belong to `type(obj)` contain a dictionary attribute `__dict__` . This situation is viuslaized in the following diagram:
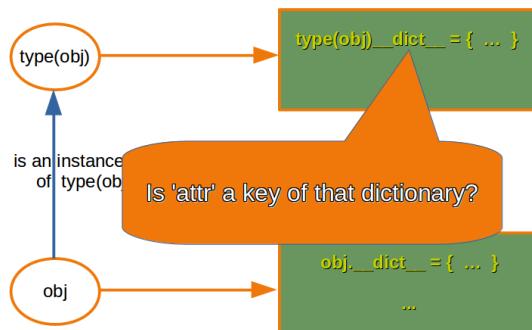
`obj.ap` has a lookup chain starting with `obj.__dict__['ap']`, i.e. checks if `obj.ap` is a key of the dictionary `obj.__dict__['ap']`.

If `ap` is not a key of `obj.__dict__`, it will try to lookup `type(obj).__dict__['ap']`.

If `obj` is not contained in this dictionary either, it will continue checking through the base classes of `type(ap)` excluding metaclasses.

We demonstrate this in an example:

```python
class A:

    ca_A = "class attribute of A"

    def __init__(self):
        self.ia_A = "instance attribute of A instance"
```

```python
class B(A):

    ca_B = "class attribute of B"

    def __init__(self):
        super().__init__()
        self.ia_B = "instance attribute of A instance"

x = B()

print(x.ia_B)
print(x.ca_B)
print(x.ia_A)
print(x.ca_A)
```

```
instance attribute of A instance
class attribute of B
instance attribute of A instance
class attribute of A
```

If we call `print(x.non_existing)` we get the following exception:

```
---------------------------------------------------------------
--------
AttributeError                              Traceback (most recent ca
ll last)
<ipython-input-4-119192d61d5e> in <module>
----> 1 print(x.non_existing)

AttributeError: 'B' object has no attribute 'non_existing'
```

If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Descriptors provide a powerful, general purpose protocol, which is the underlying mechanism behind properties, methods, static methods, class methods, and super(). Descriptors were needed to implement the so-called new style classes introduced in version 2.2. The "new style classes" are the default nowadays.

DESCRIPTOR PROTOCOL

The general descriptor protocol consists of three methods:

```
descr.__get__(self, obj, type=None) -> value
```

```
descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

If you define one or more of these methods, you will create a descriptor. We distinguish between data descriptors and non-data descriptors:

non-data descriptor
If we define only the __get__() method, we create a non-data descriptor, which are mostly used for methods.

data descriptor
If an object defines __set__() or __delete__(), it is considered a data descriptor. To make a read-only data descriptor, define both __get__() and __set__() with the __set__() raising an AttributeError when called. Defining the __set__() method with an exception raising placeholder is enough to make it a data descriptor.

We finally come now to our simple descriptor example in the following code:

```python
class SimpleDescriptor(object):
    """
    A simple data descriptor that can set and return values
    """

    def __init__(self, initval=None):
        print("__init__ of SimpleDecorator called with initval: ", initval)
        self.__set__(self, initval)

    def __get__(self, instance, owner):
        print(instance, owner)
        print('Getting (Retrieving) self.val: ', self.val)
        return self.val

    def __set__(self, instance, value):
        print('Setting self.val to ', value)
        self.val = value

class MyClass(object):

    x = SimpleDescriptor("green")
```

```
m = MyClass()
print(m.x)
m.x = "yellow"
print(m.x)
```

```
__init__ of SimpleDecorator called with initval:  green
Setting self.val to  green
<__main__.MyClass object at 0x7efe93ff4a20> <class '__main__.MyCla
ss'>
Getting (Retrieving) self.val:  green
green
Setting self.val to  yellow
<__main__.MyClass object at 0x7efe93ff4a20> <class '__main__.MyCla
ss'>
Getting (Retrieving) self.val:  yellow
yellow
```

The third parameter `owner` of `__get__` is always the owner class and provides users with an option to do something with the class that was used to call the descriptor. Usually, i.e. if the descriptor is called through an object `obj`, the object type can be deduced by calling `type(obj)`. The situation is different, if the descriptor is invoked through a class. In this case it is `None` and it wouldn't be possible to access the class unless the third argument is given. The second parameter `instance` is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the owner.

Let's have a look at the `__dict__` dictionaries of both the instances and the class:

'x' is a class attribute in the previous class. You may have asked yourself, if we could also use this mechanism in the `__init__` method to define instance attribute. This is not possible. The methods `__get__()`, `__set__()`, and `__delete__()` will only apply if an instance of the class containing the method (a so-called descriptor class) appears in an owner class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples above, the attribute 'x' is in the owner `__dict__` of the owner class MyClass, as we can see below:

```
print(m.__dict__)
print(MyClass.__dict__)
print(SimpleDescriptor.__dict__)
```

```
{}
{'__module__': '__main__', 'x': <__main__.SimpleDescriptor object
at 0x7efe93ff49e8>, '__dict__': <attribute '__dict__' of 'MyClas
s' objects>, '__weakref__': <attribute '__weakref__' of 'MyClass'
objects>, '__doc__': None}
{'__module__': '__main__', '__doc__': '\n    A simple data descrip
tor that can set and return values\n    ', '__init__': <function S
impleDescriptor.__init__ at 0x7efe93fc8d08>, '__get__': <function
SimpleDescriptor.__get__ at 0x7efe93fc8840>, '__set__': <function
SimpleDescriptor.__set__ at 0x7efe93fc88c8>, '__dict__': <attribut
e '__dict__' of 'SimpleDescriptor' objects>, '__weakref__': <attri
bute '__weakref__' of 'SimpleDescriptor' objects>}
```

```
print(MyClass.__dict__['x'])
```

```
<__main__.SimpleDescriptor object at 0x7efe93ff49e8>
```

It is possible to call a descriptor directly by its method name, e.g. `d.__get__(obj)`.

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, `obj.d` looks up `d` in the dictionary `__dict__` of `obj`. If `d` defines the method `__get__()`, then `d.__get__(obj)` is invoked according to the precedence rules listed below.

It makes a difference if `obj` is an object or a class:

- For objects, the method to control the invocation is in `object.__getattribute__()` which transforms `b.x` into the call `type(b).__dict__['x'].__get__(b, type(b))`. The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to `__getattr__()` if provided.

- For classes, the corresponding method is in the `type` class, i.e. `type.__getattribute__()` which transforms B.x into `B.__dict__['x'].__get__(None, B)`.

`__getattribute__` is not implemented in Python but in C. The following Python code is a simulation of the logic in Python. We can see that the descriptors are called by the `__getattribute__` implementations.

```python
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = type.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

```
m.__getattribute__("x")
```

```
<__main__.MyClass object at 0x7efe93ff4a20> <class '__main__.MyCla
ss'>
Getting (Retrieving) self.val:  yellow
```
Output: `'yellow'`

The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The attribute lookup `super(B, obj).m` searches `obj.__class__.__mro__` for the base class `A` immediately following `B` and then returns `A.__dict__['m'].__get__(obj, B)`. If not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

The details above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for object, type, and super(). Classes inherit this machinery when they derive from object or if they have a meta-class providing similar functionality. This means also that one can turn-off automatic descriptor calls by overriding `__getattribute__()`.

```python
from weakref import WeakKeyDictionary

class Voter:

    required_age = 18 # in Germany

    def __init__(self):
        self.age = WeakKeyDictionary()

    def __get__(self, instance_obj, objtype):
        return self.age.get(instance_obj)

    def __set__(self, instance, new_age):
        if new_age < Voter.required_age:
            msg = '{name} is not old enough to vote in Germany'
            raise Exception(msg.format(name=instance.name))
        self.age[instance] = new_age
        print('{name} can vote in Germany'.format(
            name=instance.name))

    def __delete__(self, instance):
        del self.age[instance]


class Person:
    voter_age = Voter()
```

```python
    def __init__(self, name, age):
        self.name = name
        self.voter_age = age


p1 = Person('Ben', 23)
p2 = Person('Emilia', 22)

p2.voter_age
```

```
Ben can vote in Germany
Emilia can vote in Germany
```

Output: 22

A pure sample implementation of a property() class (https://docs.python.org/3/howto/descriptor.html#properties)

```python
class Property:
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
```

```
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__do
c__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__do
c__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__do
c__)
```

A simple class using our Property implementation.

```
class A:

    def __init__(self, prop):
        self.prop = prop

    @Property
    def prop(self):
        print("The Property 'prop' will be returned now:")
        return self.__prop

    @prop.setter
    def prop(self, prop):
        print("prop will be set")
        self.__prop = prop
```

Using our class A:

```
print("Initializing the Property 'prop' with the value 'Python'")
x = A("Python")
print("The value is: ", x.prop)
print("Reassigning the Property 'prop' to 'Python descriptors'")
x.prop = "Python descriptors"
print("The value is: ", x.prop)
```

```
Initializing the Property 'prop' with the value 'Python'
prop will be set
The Property 'prop' will be returned now:
The value is:  Python
Reassigning the Property 'prop' to 'Python descriptors'
prop will be set
The Property 'prop' will be returned now:
The value is:  Python descriptors
```

Let's make our Property implementation a little bit more talkative with some print functions to see what is going on:

```python
class Property:
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        print("\n__init__ of Property called with:")
        print("fget=" + str(fget) + " fset=" + str(fset) + \
              " fdel=" + str(fdel) + " doc=" + str(doc))
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        print("\nProperty.__get__ has been called!")
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        print("\nProperty.__set__ has been called!")
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        print("\nProperty.__delete__ has been called!")
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
```

```python
            self.fdel(obj)

    def getter(self, fget):
        print("\nProperty.getter has been called!")
        return type(self)(fget, self.fset, self.fdel, self.__do
c__)

    def setter(self, fset):
        print("\nProperty.setter has been called!")
        return type(self)(self.fget, fset, self.fdel, self.__do
c__)

    def deleter(self, fdel):
        print("\nProperty.deleter has been called!")
        return type(self)(self.fget, self.fset, fdel, self.__do
c__)

class A:

    def __init__(self, prop):
        self.prop = prop

    @Property
    def prop(self):
        """ This will be the doc string of the property """
        print("The Property 'prop' will be returned now:")
        return self.__prop

    @prop.setter
    def prop(self, prop):
        print("prop will be set")
        self.__prop = prop

    def prop2_getter(self):
        return self.__prop2

    def prop2_setter(self, prop2):
        self.__prop2 = prop2

    prop2 = Property(prop2_getter, prop2_setter)

print("Initializing the Property 'prop' with the value 'Python'")
x = A("Python")
print("The value is: ", x.prop)
print("Reassigning the Property 'prop' to 'Python descriptors'")
```

```
x.prop = "Python descriptors"
print("The value is: ", x.prop)

print(A.prop.getter(x))

def new_prop_setter(self, prop):
    if prop=="foo":
        self.__prop = "foobar"
    else:
        self.__prop = prop

A.prop.setter
```

```
__init__  of Property called with:
fget=<function A.prop at 0x7efe93fff840> fset=None fdel=None doc=N
one

Property.setter has been called!

__init__  of Property called with:
fget=<function A.prop at 0x7efe93fff840> fset=<function A.prop at
0x7efe9370f840> fdel=None doc= This will be the doc string of the
property

__init__  of Property called with:
fget=<function A.prop2_getter at 0x7efe93715048> fset=<function
A.prop2_setter at 0x7efe93715158> fdel=None doc=None
Initializing the Property 'prop' with the value 'Python'

Property.__set__ has been called!
prop will be set

Property.__get__ has been called!
The Property 'prop' will be returned now:
The value is:  Python
Reassigning the Property 'prop' to 'Python descriptors'

Property.__set__ has been called!
prop will be set

Property.__get__ has been called!
The Property 'prop' will be returned now:
The value is:  Python descriptors

Property.__get__ has been called!

Property.getter has been called!

__init__  of Property called with:
fget=<__main__.A object at 0x7efe9371b080> fset=<function A.prop a
t 0x7efe9370f840> fdel=None doc= This will be the doc string of th
e property
<__main__.Property object at 0x7efe93ff4358>

Property.__get__ has been called!
```

Output:
```
<bound method Property.setter of <__main__.Property object a
t 0x7efe93729f98>>
```

```python
class Robot:
    def __init__(self, name="Marvin", city="Freiburg"):
        self.name = name
        self.city = city

    @Property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name == "hello":
            self.__name = "hi"
        else:
            self.__name = name


x = Robot("Marvin")
print(x.name)
x.name = "Eddie"
print(x.name)
```

```
__init__ of Property called with:
fget=<function Robot.name at 0x7efe93715ea0> fset=None fdel=None d
oc=None

Property.setter has been called!

__init__ of Property called with:
fget=<function Robot.name at 0x7efe93715ea0> fset=<function Robo
t.name at 0x7efe93715400> fdel=None doc=None

Property.__set__ has been called!

Property.__get__ has been called!
Marvin

Property.__set__ has been called!

Property.__get__ has been called!
Eddie
```

```python
class A:

    def a(func):
        def helper(self, x):
```

```
            return 4 * func(self, x)
        return helper

    @a
    def b(self, x):
        return x + 1

a = A()
a.b(4)
```

20

A lot of people ask, if it is possible to automatically create descriptors at runtime. This is possible as we show in the following example. On the other hand, this example is not very useful, because the getters and setters have no real functionality:

```
class DynPropertyClass(object):

    def add_property(self, attribute):
        """ add a property to the class """

        def get_attribute(self):
            """ The value for attribute 'attribute' will be retrie
ved """
            return getattr(self, "_" + type(x).__name__ + "__" + a
ttribute)

        def set_attribute(self, value):
            """ The value for attribute 'attribute' will be retrie
ved """
            #setter = lambda self, value: self.setProperty(attribu
te, value)
            setattr(self, "_" + type(x).__name__ + "__" + attribut
e, value)

        #construct property attribute and add it to the class
        setattr(type(self), attribute, property(fget=get_attribut
e,
                                                fset=set_attribut
e,
                                                doc="Auto-generate
d method"))
```

```
x = DynPropertyClass()
x.add_property('name')
x.add_property('city')
x.name = "Henry"
x.name
x.city = "Hamburg"
print(x.name, x.city)
print(x.__dict__)
```

```
Henry Hamburg
{'_DynPropertyClass__name': 'Henry', '_DynPropertyClass__city': 'H
amburg'}
```
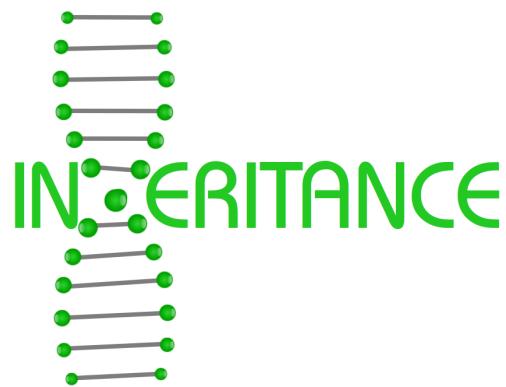
INHERITANCE
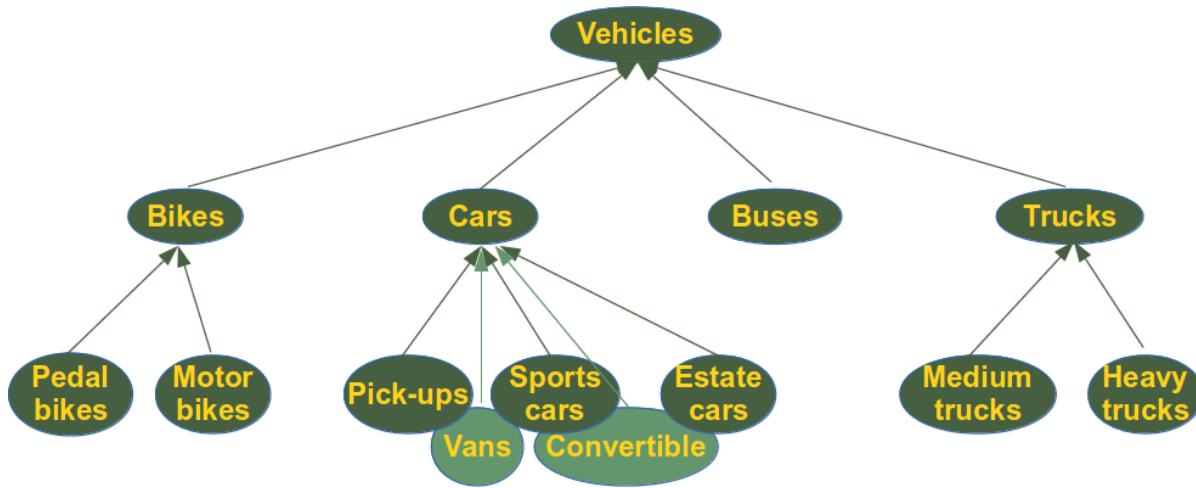
INTRODUCTION AND DEFINITIONS

No object-oriented programming language would be worthy to look at or use, if it didn't support inheritance. Inheritance was invented in 1969 for Simula. Python not only supports inheritance but multiple inheritance as well. Generally speaking, inheritance is the mechanism of deriving new classes from existing ones. By doing this, we get a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all, - though there are exceptions in some programming languages, - of the properties and behaviors of the parent object.

Inheritance allows programmers to create classes that are built upon existing classes, and this enables a class created through inheritance to inherit the attributes and methods of the parent class. This means that inheritance supports code reusability. The methods or generally speaking the software inherited by a subclass is considered to be reused in the subclass. The relationships of objects or classes through inheritance give rise to a directed graph.

The class from which a class inherits is called the parent or superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchical relationship between classes. It's similar to relationships or categorizations that we know from real life. Think about vehicles, for example. Bikes, cars, buses and trucks are vehicles. Pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

SYNTAX OF INHERITANCE IN PYTHON

The syntax for a subclass definition looks like this:

```
class DerivedClassName(BaseClassName):
    pass
```

Instead of the `pass` statement, there will be methods and attributes like in all other classes. The name BaseClassName must be defined in a scope containing the derived class definition.

Now we are ready for a simple inheritance example with Python code.

SIMPLE INHERITANCE EXAMPLE

We will stick with our beloved robots or better `Robot` class from the previous chapters of our Python tutorial to show how the principle of inheritance works. We will define a class `PhysicianRobot`, which inherits from `Robot`.

```python
class Robot:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    pass
```
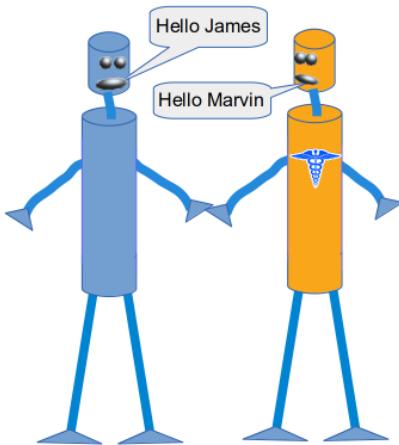
```
x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))

y.say_hi()
```

```
<__main__.Robot object at 0x7fd0080b3ba8> <class '__main__.Robot'>
<__main__.PhysicianRobot object at 0x7fd0080b3b70> <class '__mai
n__.PhysicianRobot'>
Hi, I am James
```


If you look at the code of our `PhysicianRobot` class, you can see that we haven't defined any attributes or methods in this class. As the class `PhysicianRobot` is a subclass of `Robot`, it inherits, in this case, both the method `__init__` and `say_hi`. Inheriting these methods means that we can use them as if they were defined in the `PhysicianRobot` class. When we create an instance of `PhysicianRobot`, the `__init__` function will also create a name attribute. We can apply the `say_hi` method to the `PhysisicianRobot` object `y`, as we can see in the output from the code above.

DIFFERENCE BETWEEN TYPE AND `ISINSTANCE`

You should also pay attention to the following facts, which we pointed out in other sections of our Python tutorial as well. People frequently ask where the difference between checking the type via the `type` function or the function `isinstance` is lies. The difference can be seen in the following code. We see that `isinstance` returns `True` if we compare an object either with the class it belongs to or with the superclass. Whereas the equality operator only returns `True`, if we compare an object with its own class.

```
x = Robot("Marvin")
y = PhysicianRobot("James")

print(isinstance(x, Robot), isinstance(y, Robot))
print(isinstance(x, PhysicianRobot))
print(isinstance(y, PhysicianRobot))

print(type(y) == Robot, type(y) == PhysicianRobot)
```

```
True True
False
True
False True
```

This is even true for arbitrary ancestors of the class in the inheritance line:

```
class A:
    pass

class B(A):
    pass

class C(B):
    pass

x = C()
print(isinstance(x, A))
```

```
True
```

Now it should be clear, why PEP 8, the official Style Guide for Python code, says: "Object type comparisons should always use isinstance() instead of comparing types directly."

OVERRIDING

Let us get back to our new `PhysicianRobot` class. Imagine now that an instance of a `PhysicianRobot` should say hi in a different way. In this case, we have to redefine the method `say_hi` inside of the subclass `PhysicianRobot`:

```
class Robot:

    def __init__(self, name):
        self.name = name
```

```
    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):

    def say_hi(self):
        print("Everything will be okay! ")
        print(self.name + " takes care of you!")

y = PhysicianRobot("James")
y.say_hi()
```

```
Everything will be okay!
James takes care of you!
```

What we have done in the previous example is called overriding. A method of a parent class gets overridden by simply defining a method with the same name in the child class.

If a method is overridden in a class, the original method can still be accessed, but we have to do it by calling the method directly with the class name, i.e. `Robot.say_hi(y)`. We demonstrate this in the following code:

```
y = PhysicianRobot("Doc James")
y.say_hi()
print("... and now the 'traditional' robot way of saying hi :-)")
Robot.say_hi(y)
```

```
Everything will be okay!
Doc James takes care of you!
... and now the 'traditional' robot way of saying hi :-)
Hi, I am Doc James
```

We have seen that an inherited class can inherit and override methods from the superclass. Besides this a subclass often needs additional methods with additional functionalities, which do not exist in the superclass. An instance of the `PhysicianRobot` class will need for example the method `heal` so that the physician can do a proper job. We will also add an attribute `health_level` to the `Robot` class, which can take a value between `0` and `1`. The robots will 'come to live' with a random value between `0` and `1`. If the `health_level` of a `Robot` is below 0.8, it will need a doctor. We write a method `needs_a_doctor` which returns `True` if the value is below `0.8` and `False` otherwise. The 'healing' in the `heal` method is done by setting the `health_level` to a random value between the old `health_level` and 1. This value is calculated by the uniform function of the random module.

```
import random

class Robot:
```

```python
    def __init__(self, name):
        self.name = name
        self.health_level = random.random()

    def say_hi(self):
        print("Hi, I am " + self.name)

    def needs_a_doctor(self):
        if self.health_level < 0.8:
            return True
        else:
            return False

class PhysicianRobot(Robot):

    def say_hi(self):
        print("Everything will be okay! ")
        print(self.name + " takes care of you!")


    def heal(self, robo):
        robo.health_level = random.uniform(robo.health_level, 1)
        print(robo.name + " has been healed by " + self.name +
"!")

doc = PhysicianRobot("Dr. Frankenstein")

rob_list = []
for i in range(5):
    x = Robot("Marvin" + str(i))
    if x.needs_a_doctor():
        print("health_level of " + x.name + " before healing: ",
x.health_level)
        doc.heal(x)
        print("health_level of " + x.name + " after healing: ",
x.health_level)
    rob_list.append((x.name, x.health_level))

print(rob_list)
```

```
health_level of Marvin0 before healing:  0.5562005305000016
Marvin0 has been healed by Dr. Frankenstein!
health_level of Marvin0 after healing:  0.7807651150204282
health_level of Marvin1 before healing:  0.40571527448692757
Marvin1 has been healed by Dr. Frankenstein!
health_level of Marvin1 after healing:  0.4160992532325318
health_level of Marvin2 before healing:  0.3786957462635925
Marvin2 has been healed by Dr. Frankenstein!
health_level of Marvin2 after healing:  0.5474124864506639
health_level of Marvin3 before healing:  0.6384666796845331
Marvin3 has been healed by Dr. Frankenstein!
health_level of Marvin3 after healing:  0.6986491928780778
health_level of Marvin4 before healing:  0.5983126049766974
Marvin4 has been healed by Dr. Frankenstein!
health_level of Marvin4 after healing:  0.6988801787833587
[('Marvin0', 0.7807651150204282), ('Marvin1', 0.416099253232531
8), ('Marvin2', 0.5474124864506639), ('Marvin3', 0.698649192878077
8), ('Marvin4', 0.6988801787833587)]
```

When we override a method, we sometimes want to reuse the method of the parent class and at some new stuff. To demonstrate this, we will write a new version of the PhysicianRobot. `say_hi` should return the text from the Robot class version plus the text " and I am a physician!"

```python
class PhysicianRobot(Robot):

    def say_hi(self):
        Robot.say_hi(self)
        print("and I am a physician!")


doc = PhysicianRobot("Dr. Frankenstein")
doc.say_hi()
```

```
Hi, I am Dr. Frankenstein
and I am a physician!
```

We don't want to write redundant code and therefore we called `Robot.say_hi(self)`. We could also use the `super` function:

```python
class PhysicianRobot(Robot):

    def say_hi(self):
        super().say_hi()
        print("and I am a physician!")
```

```
doc = PhysicianRobot("Dr. Frankenstein")
doc.say_hi()
```

```
Hi, I am Dr. Frankenstein
and I am a physician!
```

`super` is not realls necessary in this case. One could argue that it makes the code more maintainable, because we could change the name of the parent class, but this is seldom done anyway in existing classes. The real benefit of `super` shows when we use it with multiple inheritance.

DISTINCTION BETWEEN OVERWRITING, OVERLOADING AND OVERRIDING

OVERWRITING

If we overwrite a function, the original function will be gone. The function will be redefined. This process has nothing to do with object orientation or inheritance.

```python
def f(x):
    return x + 42

print(f(3))
# f will be overwritten (or redefined) in the following:

def f(x):
    return x + 43
print(f(3))
```

```
45
46
```

This subchapter will be only interesting for C++ and Java programmers who want to know how overloading can be accomplished in Python. Those who do not know about overloading will not miss it!

In the context of object-oriented programming, you might have heard about "overloading" as well. Even though "overloading" is not directly connected to OOP. Overloading is the ability to define a function with the same name multiple times. The definitions are different concerning the number of parameters and types of the parameters. It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters. We cannot overload functions like this in Python, but it is not necessary either.

This course is, however, not about C++ and we have so far avoided using any C++ code. We want to make an exception now, so that you can see, how overloading works in C++.

```cpp
#include
#include

using namespace std;

int successor(int number) {
    return number + 1;
}

double successor(double number) {
    return number + 1;
}

int main() {

    cout << successor(10) << endl;
    cout << successor(10.3) << endl;

    return 0;
}
```

We defined the successor function twice: One time for int and the other time with float as a Parameter. In Python the function can be defined like this, as you will know for sure:

In [ ]:

```python
def successor(x):
    return x + 1
```

As x is only a reference to an object, the Python function successor can be called with every object, even though it will create exceptions with many types. But it will work with int and float values!

Having a function with a different number of parameters is another way of function overloading. The following C++ program shows such an example. The function f can be called with either one or two integer arguments:

```cpp
 #include
using namespace std;


int f(int n);
int f(int n, int m);

int main() {
```

```cpp
        cout << "f(3): " << f(3) << endl;
        cout << "f(3, 4): " << f(3, 4) << endl;
        return 0;
    }

    int f(int n) {
        return n + 42;
    }
    int f(int n, int m) {
        return n + m + 42;
    }
```

This doesn't work in Python, as we can see in the following example. The second definition of f with two parameters redefines or overrides the first definition with one argument. Overriding means that the first definition is not available anymore.

```python
def f(n):
    return n + 42

def f(n,m):
    return n + m + 42

print(f(3, 4))
```

```
49
```

If you call f with only one parameter, you will raise an exception:

```
    f(3)


    ----------------------------------------------------------------
    --------
    TypeError                                Traceback (most recent ca
    ll last)
     in
    ----> 1 f(3)

    TypeError: f() missing 1 required positional argument: 'm'
```

Yet, it is possible to simulate the overloading behaviour of C++ in Python in this case with a default parameter:

```python
def f(n, m=None):
    if m:
        return n + m +42
    else:
        return n + 42

print(f(3), f(1, 3))
```

```
45 46
```

The * operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters:

```python
def f(*x):
    if len(x) == 1:
        return x[0] + 42
    elif len(x) == 2:
        return x[0] - x[1] + 5
    else:
        return 2 * x[0] + x[1] + 42

print(f(3), f(1, 2), f(3, 2, 1))
```
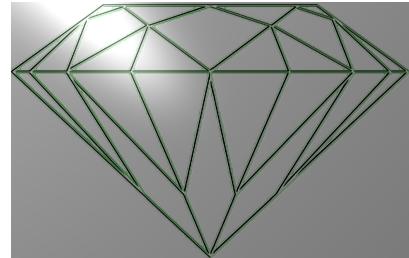
```
45 4 50
```

Overriding is already explained above!

MULTIPLE INHERITANCE

INTRODUCTION

In the previous chapter of our tutorial, we have covered inheritance, or more specific "single inheritance". As we have seen, a class inherits in this case from one class. Multiple inheritance on the other hand is a feature in which a class can inherit attributes and methods from more than one parent class. The critics point out that multiple inheritance comes along with a high level of complexity and ambiguity in situations such as the diamond problem. We will address this problem later in this chapter.
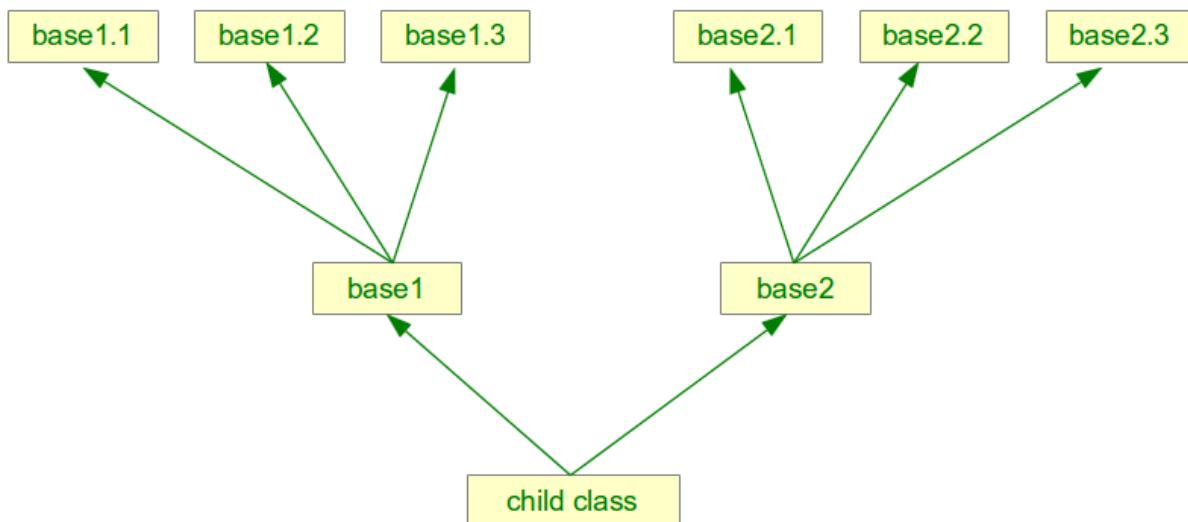
The widespread prejudice that multiple inheritance is something "dangerous" or "bad" is mostly nourished by programming languages with poorly implemented multiple inheritance mechanisms and above all by improper usage of it. Java doesn't even support multiple inheritance, while C++ supports it. Python has a sophisticated and well-designed approach to multiple inheritance.

A class definition, where a child class SubClassName inherits from the parent classes BaseClass1, BaseClass2, BaseClass3, and so on, looks like this:

```
class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):
    pass
```

It's clear that all the superclasses BaseClass1, BaseClass2, BaseClass3, ... can inherit from other superclasses as well. What we get is an inheritance tree.

EXAMPLE: CALENDARCLOCK

We want to introduce the principles of multiple inheritance with an example. For this purpose, we will implement to independent classes: a "Clock" and a "Calendar" class. After this, we will introduce a class "CalendarClock", which is, as the name implies, a combination of "Clock" and "Calendar". CalendarClock inherits both from "Clock" and "Calendar".

The class Clock simulates the tick-tack of a clock. An instance of this class contains the time, which is stored in the attributes self.hours, self.minutes and self.seconds. Principally, we could have written the `__init__` method and the set method like this:

```python
def __init__(self,hours=0, minutes=0, seconds=0):
    self._hours = hours
    self.__minutes = minutes
    self.__seconds = seconds

def set(self,hours, minutes, seconds=0):
    self._hours = hours
    self.__minutes = minutes
    self.__seconds = seconds
```

We decided against this implementation, because we added additional code for checking the plausibility of the time data into the set method. We call the set method from the `__init__` method as well, because we want to circumvent redundant code. The complete Clock class:

```python
"""
The class Clock is used to simulate a clock.
"""

class Clock(object):

    def __init__(self, hours, minutes, seconds):
        """
        The paramaters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        self.set_Clock(hours, minutes, seconds)

    def set_Clock(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 a
nd 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self.__minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between
0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self.__seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between
```

```python
0 and 59!")

    def __str__(self):
        return "{0:02d}:{1:02d}:{2:02d}".format(self._hours,
                                                self.__minutes,
                                                self.__seconds)

    def tick(self):
        """
        This method lets the clock "tick", this means that the
        internal time will be advanced by one second.

        Examples:
        >>> x = Clock(12,59,59)
        >>> print(x)
        12:59:59
        >>> x.tick()
        >>> print(x)
        13:00:00
        >>> x.tick()
        >>> print(x)
        13:00:01
        """

        if self.__seconds == 59:
            self.__seconds = 0
            if self.__minutes == 59:
                self.__minutes = 0
                if self._hours == 23:
                    self._hours = 0
                else:
                    self._hours += 1
            else:
                self.__minutes += 1
        else:
            self.__seconds += 1


if __name__ == "__main__":
    x = Clock(23,59,59)
    print(x)
    x.tick()
    print(x)
    y = str(x)
    print(type(y))
```

```
23:59:59
00:00:00
<class 'str'>
```

Let's check our exception handling by inputting floats and strings as input. We also check, what happens, if we exceed the limits of the expected values:

```
x = Clock(7.7, 45, 17)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-f91279ca09c6> in <module>
----> 1 x = Clock(7.7,45,17)

<ipython-input-1-ffb3e30af1a8> in __init__(self, hours, minutes, seconds)
     14         """
     15
---> 16         self.set_Clock(hours, minutes, seconds)
     17
     18     def set_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set_Clock(self, hours, minutes, seconds)
     28             self._hours = hours
     29         else:
---> 30             raise TypeError("Hours have to be integers between 0 and 23!")
     31         if type(minutes) == int and 0 <= minutes and minutes < 60:
     32             self.__minutes = minutes

TypeError: Hours have to be integers between 0 and 23!
```

```
x = Clock(24, 45, 17)
```

```
------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-36d0f83822b6> in <module>
----> 1 x = Clock(24, 45, 17)

<ipython-input-1-ffb3e30af1a8> in __init__(self, hours, minutes, seconds)
     14         """
     15
---> 16         self.set_Clock(hours, minutes, seconds)
     17
     18     def set_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set_Clock(self, hours, minutes, seconds)
     28             self._hours = hours
     29         else:
---> 30             raise TypeError("Hours have to be integers between 0 an
d 23!")
     31         if type(minutes) == int and 0 <= minutes and minutes < 60:
     32             self.__minutes = minutes

TypeError: Hours have to be integers between 0 and 23!
```

```
x = Clock(23, 60, 17)
```

```
------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-6c03e29741a6> in <module>
----> 1 x = Clock(23,60,17)

<ipython-input-1-ffb3e30af1a8> in __init__(self, hours, minutes, seconds)
     14         """
     15
---> 16         self.set_Clock(hours, minutes, seconds)
     17
     18     def set_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set_Clock(self, hours, minutes, seconds)
     32             self.__minutes = minutes
     33         else:
---> 34             raise TypeError("Minutes have to be integers between 0
and 59!")
     35         if type(seconds) == int and 0 <= seconds and seconds < 60:
     36             self.__seconds = seconds

TypeError: Minutes have to be integers between 0 and 59!
```

```
x = Clock("23", "60", "17")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-9d4de9fcfcdc> in <module>
----> 1 x = Clock("23", "60", "17")

<ipython-input-1-ffb3e30af1a8> in __init__(self, hours, minutes, seconds)
     14            """
     15
---> 16            self.set_Clock(hours, minutes, seconds)
     17
     18        def set_Clock(self, hours, minutes, seconds):

<ipython-input-1-ffb3e30af1a8> in set_Clock(self, hours, minutes, seconds)
     28                self._hours = hours
     29            else:
---> 30                raise TypeError("Hours have to be integers between 0 an
d 23!")
     31            if type(minutes) == int and 0 <= minutes and minutes < 60:
     32                self.__minutes = minutes

TypeError: Hours have to be integers between 0 and 23!
```

```
x = Clock(23, 17)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-dcf065c8d655> in <module>
----> 1 x = Clock(23, 17)

TypeError: __init__() missing 1 required positional argument: 'seconds'
```

We will now create a class "Calendar", which has lots of similarities to the previously defined Clock class. Instead of "tick" we have an "advance" method, which advances the date by one day, whenever it is called. Adding a day to a date is quite tricky. We have to check, if the date is the last day in a month and the number of days in the months vary. As if this isn't bad enough, we have February and the leap year problem.

The rules for calculating a leap year are the following:

- If a year is divisible by 400, it is a leap year.
- If a year is not divisible by 400 but by 100, it is not a leap year.
- A year number which is divisible by 4 but not by 100, it is a leap year.
- All other year numbers are common years, i.e. no leap years.

As a little useful gimmick, we added a possibility to output a date either in British or in American (Canadian) style.

```python
"""
The class Calendar implements a calendar.
"""


class Calendar(object):

    months = (31,28,31,30,31,30,31,31,30,31,30,31)
    date_style = "British"

    @staticmethod
    def leapyear(year):
        """
        The method leapyear returns True if the parameter year
        is a leap year, False otherwise
        """
        if not year % 4 == 0:
            return False
        elif not year % 100 == 0:
            return True
        elif not year % 400 == 0:
            return False
        else:
            return True


    def __init__(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """

        self.set_Calendar(d,m,y)


    def set_Calendar(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """

        if type(d) == int and type(m) == int and type(y) == int:
            self.__days = d
            self.__months = m
            self.__years = y
        else:
```

```python
                raise TypeError("d, m, y have to be integers!")


    def __str__(self):
        if Calendar.date_style == "British":
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__days,
                                                   self.__months,
                                                   self.__years)

        else:
            # assuming American style
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__months,
                                                   self.__days,
                                                   self.__years)


    def advance(self):
        """
        This method advances to the next date.
        """

        max_days = Calendar.months[self.__months-1]
        if self.__months == 2 and Calendar.leapyear(self.__years):
            max_days += 1
        if self.__days == max_days:
            self.__days= 1
            if self.__months == 12:
                self.__months = 1
                self.__years += 1
            else:
                self.__months += 1
        else:
            self.__days += 1


if __name__ == "__main__":
    x = Calendar(31,12,2012)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("2012 was a leapyear:")
    x = Calendar(28,2,2012)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
```

```
    x = Calendar(28,2,2013)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("1900 no leapyear: number divisible by 100 but not by 40
0: ")
    x = Calendar(28,2,1900)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("2000 was a leapyear, because number divisibe by 400: ")
    x = Calendar(28,2,2000)
    print(x, end=" ")
    x.advance()
    print("after applying advance: ", x)
    print("Switching to American date style: ")
    Calendar.date_style = "American"
    print("after applying advance: ", x)
```

```
31/12/2012 after applying advance:  01/01/2013
2012 was a leapyear:
28/02/2012 after applying advance:  29/02/2012
28/02/2013 after applying advance:  01/03/2013
1900 no leapyear: number divisible by 100 but not by 400:
28/02/1900 after applying advance:  01/03/1900
2000 was a leapyear, because number divisibe by 400:
28/02/2000 after applying advance:  29/02/2000
Switching to American date style:
after applying advance:  02/29/2000
```

At last, we will introduce our multiple inheritance example. We are now capable of implementing the originally intended class CalendarClock, which will inherit from both Clock and Calendar. The method "tick" of Clock will have to be overridden. However, the new tick method of CalendarClock has to call the tick method of Clock: Clock.tick(self)

```
"""
Module, which implements the class CalendarClock.
"""

from clock import Clock
from calendar import Calendar


class CalendarClock(Clock, Calendar):
    """
```

```python
        The class CalendarClock implements a clock with integrate
d
        calendar. It's a case of multiple inheritance, as it inher
its
        both from Clock and Calendar
    """

    def __init__(self, day, month, year, hour, minute, second):
        Clock.__init__(self,hour, minute, second)
        Calendar.__init__(self, day, month, year)


    def tick(self):
        """
        advance the clock by one second
        """
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)


if __name__ == "__main__":
    x = CalendarClock(31, 12, 2013, 23, 59, 59)
    print("One tick from ",x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(28, 2, 1900, 23, 59, 59)
    print("One tick from ",x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(28, 2, 2000, 23, 59, 59)
    print("One tick from ",x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(7, 2, 2013, 13, 55, 40)
    print("One tick from ",x, end=" ")
    x.tick()
    print("to ", x)
```
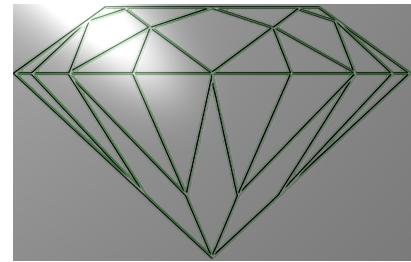
```
-----------------------------------------------------------------------
ModuleNotFoundError                         Traceback (most recent call last)
<ipython-input-1-6b533ccbb057> in <module>
      3 """
      4
----> 5 from clock import Clock
      6 from calendar import Calendar
      7

ModuleNotFoundError: No module named 'clock'
```

THE DIAMOND PROBLEM OR THE „DEADLY DIAMOND OF DEATH"

The "diamond problem" (sometimes referred as the "deadly diamond of death") is the generally used term for an ambiguity that arises when two classes B and C inherit from a superclass A, and another class D inherits from both B and C. If there is a method "m" in A that B or C (or even both of them) has overridden, and furthermore, if it does not override this method, then the question is which version of the method does D inherit? It could be the one from A, B or C.

Let's look at Python. The first Diamond Problem configuration is like this: Both B and C override the method m of A:

```python
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

If you call the method m on an instance x of D, i.e. x.m(), we will get the output "m of B called". If we transpose the order of the classes in the class header of D in "class D(C,B):", we will get the output "m of C called".

The case in which m will be overridden only in one of the classes B or C, e.g. in C:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    pass

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass

x = D()
x.m()
```

```
m of C called
```

Principially, two possibilities are imaginable: "m of C" or "m of A" could be used

We call this script with Python2.7 (python) and with Python3 (python3) to see what's happening:

```
$ python diamond1.py
m of A called
$ python3 diamond1.py
m of C called
```

Only for those who are interested in Python version2: To have the same inheritance behaviour in Python2 as in Python3, every class has to inherit from the class "object". Our class A doesn't inherit from object, so we get a so-called old-style class, if we call the script with python2. Multiple inheritance with old-style classes is governed by two rules: depth-first and then left-to-right. If you change the header line of A into "class A(object):", we will have the same behaviour in both Python versions.

SUPER AND MRO

We have seen in our previous implementation of the diamond problem, how Python "solves" the problem, i.e. in which order the base classes are browsed through. The order is defined by the so-called "Method Resolution Order" or in short MRO*.

We will extend our previous example, so that every class defines its own method m:

```
class A:
    def m(self):
        print("m of A called")
```

```python
class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    def m(self):
        print("m of D called")
```

Let's apply the method m on an instance of D. We can see that only the code of the method m of D will be executed. We can also explicitly call the methods m of the other classes via the class name, as we demonstrate in the following interactive Python session:

```python
from super1 import A,B,C,D
x = D()
B.m(x)
```

```
m of B called
```

```python
C.m(x)
```

```
m of C called
```

```python
A.m(x)
```

```
m of A called
```

Now let's assume that the method m of D should execute the code of m of B, C and A as well, when it is called. We could implement it like this:

```python
class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
        A.m(self)
```

The output is what we have been looking for:

```python
from mro import D
x = D()
```

```
x.m()
```

```
m of D called
m of B called
m of C called
m of A called
```

But it turns out once more that things are more complicated than they seem. How can we cope with the situation, if both m of B and m of C will have to call m of A as well. In this case, we have to take away the call A.m(self) from m in D. The code might look like this, but there is still a bug lurking in it:

```python
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        A.m(self)

class C(A):
    def m(self):
        print("m of C called")
        A.m(self)

class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
```

The bug is that the method m of A will be called twice:

```python
from super3 import D
x = D()
x.m()
```

```
m of D called
m of B called
m of A called
m of C called
m of A called
```

One way to solve this problem - admittedly not a Pythonic one - consists in splitting the methods m of B and C

in two methods. The first method, called `_m` consists of the specific code for B and C and the other method is still called m, but consists now of a call `self._m()` and a call `A.m(self)`. The code of the method m of D consists now of the specific code of D 'print("m of D called")', and the calls `B._m(self)`, `C._m(self)` and `A.m(self)`:

```python
class A:
    def m(self):
        print("m of A called")

class B(A):
    def _m(self):
        print("m of B called")
    def m(self):
        self._m()
        A.m(self)

class C(A):
    def _m(self):
        print("m of C called")
    def m(self):
        self._m()
        A.m(self)

class D(B,C):
    def m(self):
        print("m of D called")
        B._m(self)
        C._m(self)
        A.m(self)
```

Our problem is solved, but - as we have already mentioned - not in a pythonic way:

```python
from super4 import D
x = D()
x.m()
```

```
m of D called
m of B called
m of C called
m of A called
```

The optimal way to solve the problem, which is the "super" pythonic way, would be calling the super function:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        super().m()

class C(A):
    def m(self):
        print("m of C called")
        super().m()

class D(B,C):
    def m(self):
        print("m of D called")
        super().m()
```

It also solves our problem, but in a beautiful design as well:

```
x = D()
x.m()
```

```
m of D called
m of B called
m of C called
m of A called
```

The super function is often used when instances are initialized with the `__init__` method:

```
class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()
```

```
class D(B,C):
    def __init__(self):
        print("D.__init__")
        super().__init__()
```

We demonstrate the way of working in the following interactive session:

```
d = D()
```

```
D.__init__
B.__init__
C.__init__
A.__init__
```

```
c = C()
```

```
C.__init__
A.__init__
```

```
b = B()
```

```
B.__init__
A.__init__
```

```
a = A()
```

```
A.__init__
```

The question arises about how the super functions makes decisions. How does it decide which class has to be used? As we have already mentioned, it uses the so-called method resolution order(MRO). It is based on the "C3 superclass linearisation" algorithm. This is called a linearisation, because the tree structure is broken down into a linear order. The mro method can be used to create this list:

```
D.mro()
```

Output: `[__main__.D, __main__.B, __main__.C, __main__.A, object]`

```
B.mro()
```

Output: `[__main__.B, __main__.A, object]`

```
A.mro()
```

Output: `[__main__.A, object]`

POLYMORPHISM

Polymorphism is construed from two Greek words. "Poly" stands for "much" or "many" and "morph" means shape or form. Polymorphism is the state or condition of being polymorphous, or if we use the translations of the components "the ability to be in many shapes or forms. Polymorphism is a term used in many scientific areas. In crystallography it defines the state, if something crystallizes into two or more chemically identical but crystallographically distinct forms. Biologists know polymorphism as the existence of an organism in several form or colour varieties. The Romans even had a god, called Morpheus, who is able to take any human form: Morheus appears in Ovid's metamorphoses and is the son of Somnus, the god of sleep. You can admire Morpheus and Iris in the picture on the right side.

So, before we fall asleep, we get back to Python and to what polymorphism means in the context of programming languages. Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms. We can have in some programming languages polymorphic functions or methods, for example. Polymorphic functions or methods can be applied to arguments of different types, and they can behave differently depending on the type of the arguments to which they are applied. We can also define the same function name with a varying number of parameter.

Let's have a look at the following Python function:

```python
def f(x, y):
    print("values: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
f(42.0, 43.9)
```

```
values:  42 43
values:  42 43.7
values:  42.3 43
values:  42.0 43.9
```

We can call this function with various types, as demonstrated in the example. In typed programming languages

---

like Java or C++, we would have to overload f to implement the various type combinations.

Our example could be implemented like this in C++:

```cpp
#include
using namespace std;

void f(int x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(int x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(double x, int y ) {
    cout << "values: " << x << ", " << x << endl;
}

void f(double x, double y ) {
    cout << "values: " << x << ", " << x << endl;
}


int main()
{
    f(42, 43);
    f(42, 43.7);
    f(42.3,43);
    f(42.0, 43.9);
}
```

Python is implicitly polymorphic. We can apply our previously defined function f even to lists, strings or other types, which can be printed:

```python
def f(x,y):
    print("values: ", x, y)
```

```python
f([3,5,6],(3,5))
```

```
values:  [3, 5, 6] (3, 5)
```

```python
f("A String", ("A tuple", "with Strings"))
```

```
values:  A String ('A tuple', 'with Strings')
```

```
f({2,3,9}, {"a":3.4,"b":7.8, "c":9.04})
```

```
values:  {9, 2, 3} {'a': 3.4, 'b': 7.8, 'c': 9.04}
```

FOOTNOTES

* Python has used since 2.3 the „C3 superclass linearisation"-algorithm to determine the MRO. -->

MULTIPLE INHERITANCE: EXAMPLE

ROBOT CLASSES

This chapter of our tutorial is meant to deepen the understanding of multiple inheritance that the reader has built up in our previous chapter. We will provide a further extentive example for this important object oriented principle of the programming language Python. We will use a variation of our Robot class as the superclass. We will also summarize some other important aspects of object orientation with Python like properties. We will also work out the differences between overwriting, overloading and overriding.

This example has grown during my onsite Python training classes, because I urgently needed simple and easy to understand examples of subclassing and above all one for multiple inheritance.
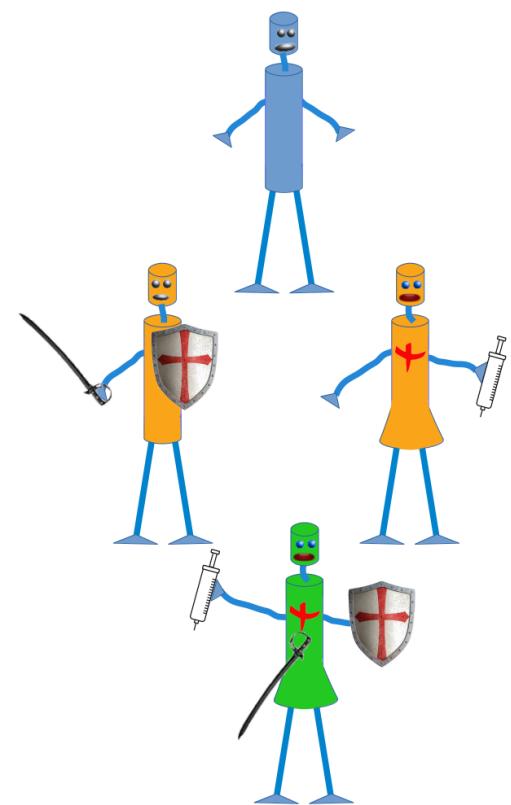
Starting from the superclass `Robot` we will derive two classes: A `FightingRobot` class and a `NursingRobot` class.

Finally we will define a 'combination' of both the `FightingRobot` class and the `NursingRobot` class, i.e. we will implement a class `FightingNurseRobot`, which will inherit both from `FightingRobot` and `NursingRobot`.

Let us start with our `Robot` class: We use a private class attribute `__illegal_names` containing a set of names not allowed to be used for naming robots.

By providing an `__add__` method we make sure that our robots are capable of propagating. The name of the resulting robot will be automatically created. The name of a 'baby' robot will consist of the concatenation of the names of both parents separated by an hyphen. If a parent name has a name containing a hyphen, we will use only the first part before the hyphen.

The robots will 'come to live' with a random value between 0 and 1 for the attribute `health_level`. If the health_level of a Robot is below a threshold, which is defined by the class attribute `Robot.__crucial_health_level`, it will need the nursing powers of a robot from the `NursingClass`. To determine if a Robots needs healing, we provide a method `needs_a_nurse` which

returns True if the value is below `Robot.__crucial_health_level` and False otherwise.

```python
import random

class Robot():

    __illegal_names = {"Henry", "Oscar"}
    __crucial_health_level = 0.6

    def __init__(self, name):
        self.name = name  #---> property setter
        self.health_level = random.random()

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name in Robot.__illegal_names:
            self.__name = "Marvin"
        else:
            self.__name = name

    def __str__(self):
        return self.name + ", Robot"

    def __add__(self, other):
        first = self.name.split("-")[0]
        second = other.name.split("-")[0]
        return Robot(first + "-" + second)

    def needs_a_nurse(self):
        if self.health_level < Robot.__crucial_health_level:
            return True
        else:
            return False

    def say_hi(self):
        print("Hi, I am " + self.name)
        print("My health level is: " + str(self.health_level))
```

We can test the newly designed `Robot` class now. Watch out how the hyphened names change from

generation to generation:

```
first_generation = (Robot("Marvin"),
                    Robot("Enigma-Alan"),
                    Robot("Charles-Henry"))

gen1 = first_generation # used as an abbreviation
babies = [gen1[0] + gen1[1], gen1[1] + gen1[2]]
babies.append(babies[0] + babies[1])
for baby in babies:
    baby.say_hi()
```

```
Hi, I am Marvin-Enigma
My health level is: 0.8879034405855623
Hi, I am Enigma-Charles
My health level is: 0.46930784534193803
Hi, I am Marvin-Enigma
My health level is: 0.2009467867758803
```

SUBCLASS NURSINGROBOT

We are ready now for subclassing the `Robot` class. We will start by creating the `NursingRobot` class. We extend the `__init__` method with a new attribute `healing_power`. At first we have to understand the concept of 'healing power'. Generally, it makes only sense to heal a `Robot`, if its health level is below 1. The 'healing' in the heal method is done by setting the health_level to a random value between the old health_level and `healing_power` of a `NursingRobot`. This value is calculated by the uniform function of the random module.

```
class NursingRobot(Robot):

    def __init__(self, name="Hubert", healing_power=None):
        super().__init__(name)
        if healing_power is None:
            self.healing_power = random.uniform(0.8, 1)
        else:
            self.healing_power = healing_power

    def say_hi(self):
        print("Well, well, everything will be fine ... " + self.na
me + " takes care of you!")
```

```python
    def say_hi_to_doc(self):
        Robot.say_hi(self)

    def heal(self, robo):
        if robo.health_level > self.healing_power:
            print(self.name + " not strong enough to heal " + robo.name)
        else:
            robo.health_level = random.uniform(robo.health_level, self.healing_power)
            print(robo.name + " has been healed by " + self.name + "!")
```

Let's heal the robot class instances which we created so far. If you look at the code, you may wonder about the function `chain`, which is a generator from the `itertools` module. Logically, the same thing happens, as if we had used `first_generation + babies`, but `chain` is not creating a new list. `chain` is iterating over both lists, one after the other and this is efficient!

```python
from itertools import chain
nurses = [NursingRobot("Hubert"),
          NursingRobot("Emma", healing_power=1)]

for nurse in nurses:
    print("Healing power of " + nurse.name,
          nurse.healing_power)

print("\nLet's start the healing")
for robo in chain(first_generation, babies):
    robo.say_hi()
    if robo.needs_a_nurse():
        # choose randomly a nurse:
        nurse = random.choice(nurses)
        nurse.heal(robo)
        print("New health level: ", robo.health_level)
    else:
        print(robo.name + " is healthy enough!")
    print()
```

```
Healing power of Hubert 0.8730202054681965
Healing power of Emma 1

Let's start the healing
Hi, I am Marvin
My health level is: 0.3529060178554311
Marvin has been healed by Emma!
New health level:  0.9005591116354148

Hi, I am Enigma-Alan
My health level is: 0.0004876210374201717
Enigma-Alan has been healed by Emma!
New health level:  0.5750360827887102

Hi, I am Charles-Henry
My health level is: 0.06446896431387994
Charles-Henry has been healed by Hubert!
New health level:  0.09474181073613296

Hi, I am Marvin-Enigma
My health level is: 0.8879034405855623
Marvin-Enigma is healthy enough!

Hi, I am Enigma-Charles
My health level is: 0.46930784534193803
Enigma-Charles has been healed by Hubert!
New health level:  0.6182358513871242

Hi, I am Marvin-Enigma
My health level is: 0.2009467867758803
Marvin-Enigma has been healed by Emma!
New health level:  0.8204101216553784
```

An interesting question is, what would happen, if Hubert and Emma get added. The question is, what the resulting type will be:

```
x = nurses[0] + nurses[1]
x.say_hi()
print(type(x))
```

```
Hi, I am Hubert-Emma
My health level is: 0.8844357538670625
<class '__main__.Robot'>
```

We see that the result of addition of two nursing robots is a plain robot of type `Robot` . This is not wrong but bad design. We want the resulting robots to be an instance of the `NursingRobot` class of course. One way to fix this would be to overload the `__add__` method inside of the `NursingRobot` class:

```
def __add__(self, other):
    first = self.name.split("-")[0]
    second = other.name.split("-")[0]
    return NursingRobot(first + "-" + second)
```

This is also bad design, because it is mainly a copy the original function with the only exception of creating an instance of NursingRobot instead of a Robot instance. An elegant solution would be having `__add__` more generally defined. Instead of creating always a `Robot` instance, we could have made it dependent on the type of self by using `type(self)` . For simplicity's sake we repeat the complete example:

```python
import random

class Robot():

    __illegal_names = {"Henry", "Oscar"}
    __crucial_health_level = 0.6

    def __init__(self, name):
        self.name = name   #---> property setter
        self.health_level = random.random()

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name in Robot.__illegal_names:
            self.__name = "Marvin"
        else:
            self.__name = name

    def __str__(self):
        return self.name + ", Robot"

    def __add__(self, other):
        first = self.name.split("-")[0]
        second = other.name.split("-")[0]
        return type(self)(first + "-" + second)

    def needs_a_nurse(self):
```

```python
        if self.health_level < Robot.__crucial_health_level:
            return True
        else:
            return False


    def say_hi(self):
        print("Hi, I am " + self.name)
        print("My health level is: " + str(self.health_level))


class NursingRobot(Robot):

    def __init__(self, name="Hubert", healing_power=None):
        super().__init__(name)
        if healing_power:
            self.healing_power = healing_power
        else:
            self.healing_power = random.uniform(0.8, 1)

    def say_hi(self):
        print("Well, well, everything will be fine ... " + self.na
me + " takes care of you!")


    def say_hi_to_doc(self):
        Robot.say_hi(self)

    def heal(self, robo):
        if robo.health_level > self.healing_power:
            print(self.name + " not strong enough to heal " + rob
o.name)
        else:
            robo.health_level = random.uniform(robo.health_level,
self.healing_power)
            print(robo.name + " has been healed by " + self.name
+ "!")
```

SUBCLASS FIGHTINGROBOT

Unfortunately, our virtual robot world is not better than their human counterpart. In other words, there will be some fighting going on as well. We subclass `Robot` once again to create a class with the name `FightingRobot`.

```python
class FightingRobot(Robot):
```

```
    __maximum_damage = 0.2

    def __init__(self, name="Hubert",
                 fighting_power=None):
        super().__init__(name)
        if fighting_power:
            self.fighting_power = fighting_power
        else:
            max_dam = FightingRobot.__maximum_damage
            self.fighting_power = random.uniform(max_dam, 1)


    def say_hi(self):
        print("I am the terrible ... " + self.name)

    def attack(self, other):
        other.health_level = \
                    other.health_level * self.fighting_power
        if isinstance(other, FightingRobot):
            # the other robot fights back
            self.health_level = \
                    self.health_level * other.fighting_power
```

Let us see now, how the fighting works:

```
fighters = (FightingRobot("Rambo", 0.4),
            FightingRobot("Terminator", 0.2))

for robo in first_generation:
    print(robo, robo.health_level)
    fighters[0].attack(robo)
    print(robo, robo.health_level)
```

```
Marvin, Robot 0.9005591116354148
Marvin, Robot 0.36022364465416595
Enigma-Alan, Robot 0.5750360827887102
Enigma-Alan, Robot 0.23001443311548408
Charles-Henry, Robot 0.09474181073613296
Charles-Henry, Robot 0.037896724294453184
```

What about Rambo fighting Terminator? This spectacular fight can be viewed in the following code:

```
# let us make them healthier first:
```

```
print("Before the battle:")
for fighter in fighters:
    nurses[1].heal(fighter)
    print(fighter,
          fighter.health_level,
          fighter.fighting_power)

fighters[0].attack(fighters[1])

print("\nAfter the battle:")
for fighter in fighters:
    print(fighter,
          fighter.health_level,
          fighter.fighting_power)
```

```
Before the battle:
Rambo has been healed by Emma!
Rambo, Robot 0.8481669878334196 0.4
Terminator has been healed by Emma!
Terminator, Robot 0.8458447223409832 0.2

After the battle:
Rambo, Robot 0.16963339756668394 0.4
Terminator, Robot 0.33833788893639327 0.2
```

AN EXAMPLE OF MULTIPLE INHERITANCE

The underlying idea of the following class `FightingNurseRobot` consists in having robots who can both heal and fight.

```
class FightingNurseRobot(NursingRobot, FightingRobot):

    def __init__(self, name, mode="nursing"):
        super().__init__(name)
        self.mode = mode     # alternatively "fighting"

    def say_hi(self):
        if self.mode == "fighting":
            FightingRobot.say_hi(self)
        elif self.mode == "nursing":
            NursingRobot.say_hi(self)
        else:
            Robot.say_hi(self)
```

We will instantiate two instances of `FightingNurseRobot`. You can see that after creation they are capable to heal themselves if necessary. They can also attack other robots.

```
fn1 = FightingNurseRobot("Donald", mode="fighting")
fn2 = FightingNurseRobot("Angela")

if fn1.needs_a_nurse():
    fn1.heal(fn1)
if fn2.needs_a_nurse():
    fn2.heal(fn2)
print(fn1.health_level, fn2.health_level)

fn1.say_hi()
fn2.say_hi()
fn1.attack(fn2)
print(fn1.health_level, fn2.health_level)
```

```
Angela has been healed by Angela!
0.8198049162122293 0.21016386954814018
I am the terrible ... Donald
Well, well, everything will be fine ... Angela takes care of you!
0.7176570622156614 0.16365553164488025
```

MAGIC METHODS AND OPERATOR OVERLOADING

INTRODUCTION

The so-called magic methods have nothing to do with wizardry. You have already seen them in the previous chapters of our tutorial. They are special methods with fixed names. They are the methods with this clumsy syntax, i.e. the double underscores at the beginning and the end. They are also hard to talk about. How do you pronounce or say a method name like `__init__`? "Underscore underscore init underscore underscore" sounds horrible and is almost a tongue twister. "Double underscore init double underscore" is a lot better, but the ideal way is "dunder init dunder" *That's why magic methods methods are sometimes called dunder methods*!

So what's magic about the `__init__` method? The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance x of a class A with the statement "x = A()", Python will do the necessary calls to `__new__` and `__init__`.

`__init__`
`__add__`
`__sub__`
`__mul__`
`__floordiv__`
`__mod__`
`__pow__`
`__lshift__`
`__rshift__`
`__and__`

Towards the end of this chapter of our tutorial we will introduce the `__call__` method. It is overlooked by many beginners and even advanced programmers of Python. It is a functionality which many programming languages do not have, so programmers generally do not look for it. The `__call__` method enables Python programmers to write classes where the instances behave like functions. Both functions and the instances of such classes are called callables.

We have encountered the concept of operator overloading many times in the course of this tutorial. We had used the plus sign to add numerical values, to concatenate strings or to combine lists:

```
4 + 5
```

Output: 9

```
3.8 + 9
```

Output: 12.8
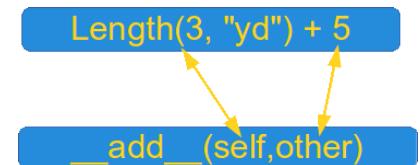
```
"Peter" + " " + "Pan"
```
Output: 'Peter Pan'

```
[3,6,8] + [7,11,13]
```
Output: [3, 6, 8, 7, 11, 13]

It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the `__add__` method. For "-" it is `__sub__` and so on. We have a complete listing of all the magic methods a little further down.

The mechanism works like this: If we have an expression "x + y" and x is an instance of class K, then Python will check the class definition of K. If K has a method `__add__` it will be called with `x.__add__(y)`, otherwise we will get an error message:

```
Traceback (most recent call last):
  File "", line 1, in
TypeError: unsupported operand type(s) for +: 'K' and 'K'
```

OVERVIEW OF MAGIC METHODS

BINARY OPERATORS

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)

Operator	Method
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

EXAMPLE CLASS: LENGTH

We will demonstrate the Length class and how you can overload the "+" operator for your own class. To do this, we have to overload the `__add__` method. Our class contains the `__str__` and `__repr__` methods as well. The instances of the class Length contain length or distance information. The attributes of an instance are self.value and self.unit.

This class allows us to calculate expressions with mixed units like this one:

```
2.56 m + 3 yd + 7.8 in + 7.03 cm
```

The class can be used like this:

```python
from unit_conversions import Length
L = Length
print(L(2.56,"m") + L(3,"yd") + L(7.8,"in") + L(7.03,"cm"))
```

```
5.57162
```

The listing of the class:

```python
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit +
"')"
```

```
if __name__ == "__main__":
    x = Length(4)
    print(x)
    y = eval(repr(x))

    z = Length(4.5, "yd") + Length(1)
    print(repr(z))
    print(z)
```

```
4
Length(5.593613298337708, 'yd')
5.1148
```

We use the method `__iadd__` to implement the extended assignment:

```
def __iadd__(self, other):
    l = self.Converse2Metres() + other.Converse2Metres()
    self.value = l / Length.__metric[self.unit]
    return self
```

Now we are capable of writing the following assignments:

```
x += Length(1)
x += Length(4, "yd")
```

We added 1 metre in the example above by writing "x += Length(1))". Most certainly, you will agree with us that it would be more convenient to simply write "x += 1" instead. We also want to treat expressions like "Length(5,"yd") + 4.8" similarly. So, if somebody uses a type int or float, our class takes it automatically for "metre" and converts it into a Length object. It's easy to adapt our `__add__` and `__iadd__` method for this task. All we have to do is to check the type of the parameter "other":

```
def __add__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit )

def __iadd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    self.value = l / Length.__metric[self.unit]
```

```
        return self
```

It's a safe bet that if somebody works with adding integers and floats from the right side for a while, he or she will want to have the same from the left side! SWhat will happen, if we execute the following code line:

```
x = 5 + Length(3, "yd")
```

We will get an exception:

```
AttributeError: 'int' object has no attribute 'Converse2Metres'
```

Of course, the left side has to be of type "Length", because otherwise Python tries to apply the `__add__` method from int, which can't cope with Length objects as second arguments!

Python provides a solution for this problem as well. It's the `__radd__` method. It works like this: Python tries to evaluate the expression "5 + Length(3, 'yd')". First it calls int. `__add__` (5,Length(3, 'yd')), which will raise an exception. After this it will try to invoke Length. `__radd__` (Length(3, "yd"), 5). It's easy to recognize that the implementation of `__radd__` is analogue to `__add__`:

```python
def __radd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit )
```

It's advisable to make use of the `__add__` method in the `__radd__` method:

```python
def __radd__(self, other):
    return Length.__add__(self,other)
```

The following diagram illustrates the relationship between `__add__` and `__radd__`:

STANDARD CLASSES AS BASE CLASSES

It's possible to use standard classes - like int, float, dict or lists - as base classes as well.

We extend the list class by adding a push method:

```python
class Plist(list):

    def __init__(self, l):
        list.__init__(self, l)

    def push(self, item):
        self.append(item)


if __name__ == "__main__":
    x = Plist([3,4])
    x.push(47)
    print(x)
```

```
[3, 4, 47]
```

This means that all the previously introduced binary and extended assignment operators exist in the "reversed" version as well:

__radd__ , __rsub__ , __rmul__ etc.

EXERCISES

Write a class with the name Ccy, similar to the previously defined Length class. Ccy should contain values in various currencies, e.g. "EUR", "GBP" or "USD". An instance should contain the amount and the currency unit. The class, you are going to design as an exercise, might be best described with the following example session:

```python
from currencies import Ccy
v1 = Ccy(23.43, "EUR")
v2 = Ccy(19.97, "USD")
print(v1 + v2)
print(v2 + v1)
print(v1 + 3) # an int or a float is considered to be a EUR value
print(3 + v1)
```

SOLUTIONS TO OUR EXERCISES

```python
"""

    The class "Ccy" can be used to define money values in various
currencies. A Ccy instance has the string attributes 'unit' (e.g.
'CHF', 'CAD' od 'EUR' and the 'value' as a float.
    A currency object consists of a value and the corresponding un
it.
    """



class Ccy:

    currencies =  {'CHF': 1.0821202355817312,
                   'CAD': 1.488609845538393,
                   'GBP': 0.8916546282920325,
                   'JPY': 114.38826536281809,
                   'EUR': 1.0,
                   'USD': 1.11123458162018}

    def __init__(self, value, unit="EUR"):
        self.value = value
        self.unit = unit

    def __str__(self):
        return "{0:5.2f}".format(self.value) + " " + self.unit

    def changeTo(self, new_unit):
        """
            An Ccy object is transformed from the unit "self.uni
t" to "new_unit"
        """
        self.value = (self.value / Ccy.currencies[self.unit] * Cc
y.currencies[new_unit])
        self.unit = new_unit

    def __add__(self, other):
        """
            Defines the '+' operator.
            If other is a CCy object the currency values
```

```
                are added and the result will be the unit of
                self. If other is an int or a float, other will
                be treated as a Euro value.
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] * Ccy.cu
rrencies[self.unit])
        return Ccy(x + self.value, self.unit)


    def __iadd__(self, other):
        """
            Similar to __add__
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] * Ccy.cu
rrencies[self.unit])
        self.value += x
        return self

    def __radd__(self, other):
        res = self + other
        if self.unit != "EUR":
            res.changeTo("EUR")
        return res

        # __sub__, __isub__ and __rsub__ can be defined analogue
```

Overwriting currencies.py

```
from currencies import Ccy

x = Ccy(10,"USD")
y = Ccy(11)
z = Ccy(12.34, "JPY")
z = 7.8 + x + y + 255 + z
print(z)

lst = [Ccy(10,"USD"), Ccy(11), Ccy(12.34, "JPY"), Ccy(12.34, "CA
D")]
```

```
z = sum(lst)

print(z)
```

```
282.91 EUR
28.40 EUR
```

Another interesting aspect of this currency converter class in Python can be shown, if we add multiplication. You will easily understand that it makes no sense to allow expressions like `"12.4 € * 3.4 USD"` (or in prefix notation: "€ 12.4 $ 3.4"), *but it makes perfect sense to evaluate "3 4.54 €"*. You can find the new currency converter class with the newly added methods for `__mul__` , `__imul__` and `__rmul__` in the following listing:

```python
"""
    The class "Ccy" can be used to define money values in various
currencies. A Ccy instance has the string attributes 'unit' (e.g.
'CHF', 'CAD' od 'EUR' and the 'value' as a float.
    A currency object consists of a value and the corresponding un
it.
"""

class Ccy:

    currencies =  {'CHF': 1.0821202355817312,
                   'CAD': 1.488609845538393,
                   'GBP': 0.8916546282920325,
                   'JPY': 114.38826536281809,
                   'EUR': 1.0,
                   'USD': 1.11123458162018}

    def __init__(self, value, unit="EUR"):
        self.value = value
        self.unit = unit

    def __str__(self):
        return "{0:5.2f}".format(self.value) + " " + self.unit

    def __repr__(self):
        return 'Ccy(' + str(self.value) + ', "' + self.unit + '")'

    def changeTo(self, new_unit):
        """
            An Ccy object is transformed from the unit "self.uni
t" to "new_unit"
        """
```

```python
        self.value = (self.value / Ccy.currencies[self.unit] * Cc
y.currencies[new_unit])
        self.unit = new_unit

    def __add__(self, other):
        """
            Defines the '+' operator.
            If other is a CCy object the currency values
            are added and the result will be the unit of
            self. If other is an int or a float, other will
            be treated as a Euro value.
        """
        if type(other) == int or type(other) == float:
                x = (other * Ccy.currencies[self.unit])
        else:
                x = (other.value / Ccy.currencies[other.unit] * Cc
y.currencies[self.unit])
        return Ccy(x + self.value, self.unit)


    def __iadd__(self, other):
        """
            Similar to __add__
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] * Ccy.cu
rrencies[self.unit])
            self.value += x
        return self

    def __radd__(self, other):
        res = self + other
        if self.unit != "EUR":
            res.changeTo("EUR")
        return res

        # __sub__, __isub__ and __rsub__ can be defined analogue


    def __mul__(self, other):
        """
            Multiplication is only defined as a scalar multiplicat
ion,
```

```
            i.e. a money value can be multiplied by an int or a fl
oat.
            It is not possible to multiply to money values
        """
        if type(other)==int or type(other)==float:
            return Ccy(self.value * other, self.unit)
        else:
            raise TypeError("unsupported operand type(s) for *: 'C
cy' and " + type(other).__name__)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __imul__(self, other):
        if type(other)==int or type(other)==float:
            self.value *= other
            return self
        else:
            raise TypeError("unsupported operand type(s) for *: 'C
cy' and " + type(other).__name__)
```

```
Overwriting currency_converter.py
```

Assuming that you have saved the class under the name currency_converter, you can use it in the following way in the command shell:

```python
from currency_converter import Ccy
x = Ccy(10.00, "EUR")
y = Ccy(10.00, "GBP")
x + y
```

Output: `Ccy(21.215104685942173, "EUR")`

```python
print(x + y)
```

```
21.22 EUR
```

```python
print(2*x + y*0.9)
```

```
30.09 EUR
```

FOOTNOTES

- as suggested by Mark Jackson
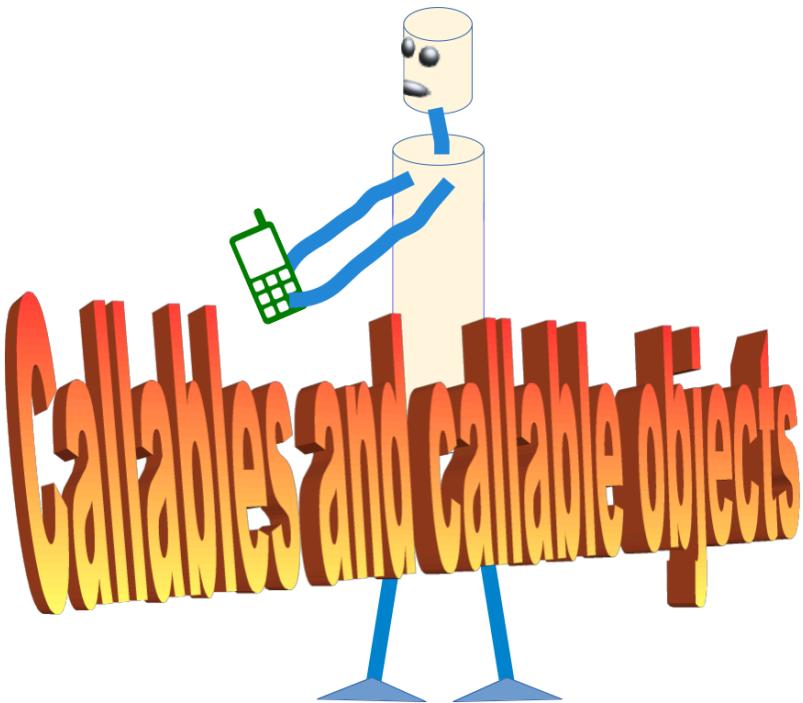
CALLABLES AND CALLABLE INSTANCES

THE CALL METHOD

There will be hardly any Python user who hasn't stumbled upon exceptions like `'dict' object is not callable` or `'int' object is not callable`. After a while they find out the reason. They used parentheses (round bracket) in situation where they shouldn't have done it. Expressions like `f(x)`, `gcd(x, y)` or `sin(3.4)` are usually used for function calls. The question is, why do we get the message `'dict' object is not callable` if we write `d('a')`, if `d` is a dictionary? Why doesn't it say `'dict' object is not a function`? First of all, when we invoke a function in Python, we also say we 'call the function'. Secondly, there are objects in Python, which are 'called' like functions but are not functions strictly speaking. There are 'lambda functions', which are defined in a different way. It is also possible to define classes, where the instances are callable like 'regular' functions. This will be achieved by adding another [magic method](#) the `__call__` method.

Before we will come to the `__call__` method, we have to know what a callable is. In general, a "callable" is an object that can be called like a function and behaves like one. All functions are also callables. Python provides a function with the name `callable`. With the help of this funciton we can determine whether an object is callable or not. The function `callable` returns a Boolean truth value which indicates whether the object passed as an argument can be called like a function or not. In addition to functions, we have already seen another form of `callables` : classes

```python
def the_answer(question):
    return 42

print("the_answer: ", callable(the_answer))
```

```
the_answer:  True
```

The `__call__` method can be used to turn the instances of the class into callables. Functions are callable objects. A callable object is an object which can be used and behaves like a function but might not be a function. By using the `__call__` method it is possible to define classes in a way that the instances will be callable objects. The `__call__` method is called, if the instance is called "like a function", i.e. using brackets. The following class definition is the simplest possible way to define a class with a `__call__` method.

```python
class FoodSupply:

    def __call__(self):
        return "spam"

foo = FoodSupply()
bar = FoodSupply()

print(foo(), bar())
```

```
spam spam
```

The previous class example is extremely simple, but useless in practical terms. Whenever we create an instance of the class, we get a callable. These callables are always defining the same constant function. A function without any input and a constant output "spam". We'll now define a class which is slightly more useful. Let us slightly improve this example:

```python
class FoodSupply:

    def __init__(self, *incredients):
        self.incredients = incredients

    def __call__(self):
        result = " ".join(self.incredients) + " plus delicious spa
m!"
        return result

f = FoodSupply("fish", "rice")
f()
```

Output: `'fish rice plus delicious spam!'`

Let's create another function:

```python
g = FoodSupply("vegetables")
g()
```

`'vegetables plus delicious spam!'`

Now, we define a class with the name TriangleArea. This class has only one method, which is the `__call__` method. The `__call__` method calculates the area of an arbitrary triangle, if the length of the three sides are given.

```python
class TriangleArea:

    def __call__(self, a, b, c):
        p = (a + b + c) / 2
        result = (p * (p - a) * (p - b) * (p - c)) ** 0.5
        return result


area = TriangleArea()

print(area(3, 4, 5))
```

```
6.0
```

This program returns 6.0. This class is not very exciting, even though we can create an arbitrary number of instances where each instance just executes an unaltered `__call__` function of the TrianlgeClass. We cannot pass parameters to the instanciation and the `__call__` of each instance returns the value of the area of the triangle. So each instance behaves like the area function.

After the two very didactic and not very practical examples, we want to demonstrate a more practical example with the following. We define a class that can be used to define linear equations:

```python
class StraightLines():

    def __init__(self, m, c):
        self.slope = m
        self.y_intercept = c

    def __call__(self, x):
        return self.slope * x + self.y_intercept

line = StraightLines(0.4, 3)

for x in range(-5, 6):
    print(x, line(x))
```
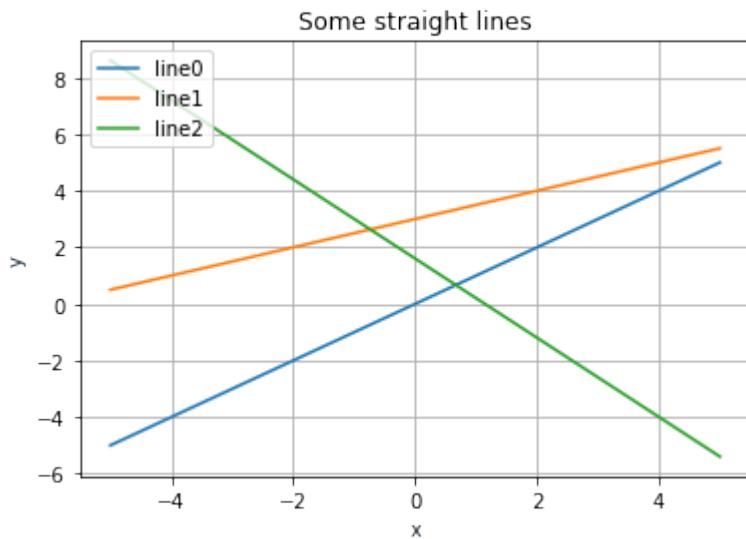
```
-5 1.0
-4 1.4
-3 1.7999999999999998
-2 2.2
-1 2.6
0 3.0
1 3.4
2 3.8
3 4.2
4 4.6
5 5.0
```

We will use this class now to create some straight lines and visualize them with matplotlib:

```python
lines = []
lines.append(StraightLines(1, 0))
lines.append(StraightLines(0.5, 3))
lines.append(StraightLines(-1.4, 1.6))

import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-5,5,100)
for index, line in enumerate(lines):
    line = np.vectorize(line)
    plt.plot(X, line(X), label='line' + str(index))

plt.title('Some straight lines')
plt.xlabel('x', color='#1C2833')
plt.ylabel('y', color='#1C2833')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```

Some straight lines

Our next example is also exciting. The class FuzzyTriangleArea defines a `__call__` method which implements a fuzzy behaviour in the calculations of the area. The result should be correct with a likelihood of p, e.g. 0.8. If the result is not correct the result will be in a range of ± v %. e.g. 0.1.

```python
import random

class FuzzyTriangleArea:

    def __init__(self, p=0.8, v=0.1):
        self.p, self.v = p, v

    def __call__(self, a, b, c):
        p = (a + b + c) / 2
        result = (p * (p - a) * (p - b) * (p - c)) ** 0.5
        if random.random() <= self.p:
            return result
        else:
            return random.uniform(result-self.v,
                                  result+self.v)

area1 = FuzzyTriangleArea()
area2 = FuzzyTriangleArea(0.5, 0.2)
for i in range(12):
    print(f"{area1(3, 4, 5):4.3f}, {area2(3, 4, 5):4.2f}")
```
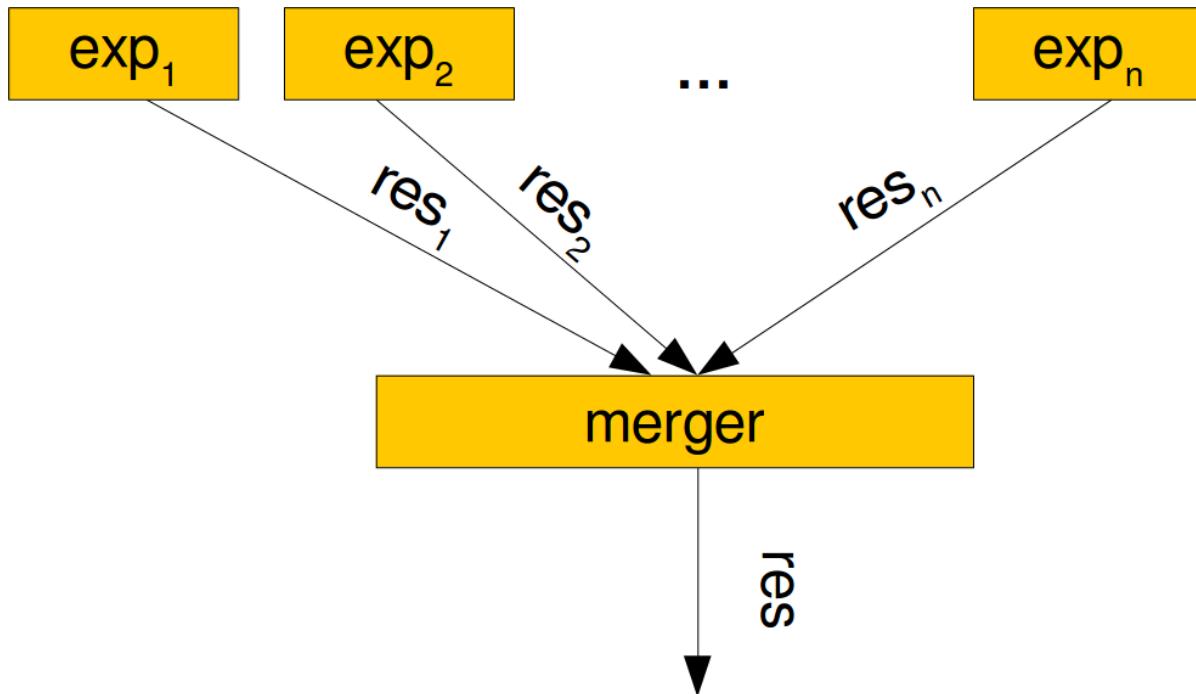
```
5.993, 5.95
6.000, 6.00
6.000, 6.00
5.984, 5.91
6.000, 6.00
6.000, 6.00
6.000, 6.17
6.000, 6.13
6.000, 6.01
5.951, 6.00
6.000, 5.95
5.963, 6.00
```

Beware that this output differs with every call! We can see the in most cases we get the right value for the area but sometimes not.

We can create many different instances of the previous class. Each of these behaves like an area function, which returns a value for the area, which may or may not be correct, depending on the instantiation parameters p and v. We can see those instances as experts (expert functions) which return in most cases the correct answer, if we use p values close to 1. If the value v is close to zero, the error will be small, if at all. The next task would be merging such experts, let's call them exp1, exp2, ..., expn to get an improved result. We can perform a vote on the results, i.e. we will return the value which is most often occuring, the correct value. Alternatively, we can calculate the arithmetic mean. We will implement both possibilities in our class FuzzyTriangleArea:

```python
from random import uniform, random
from collections import Counter

class FuzzyTriangleArea:

    def __init__(self, p=0.8, v=0.1):
        self.p, self.v = p, v

    def __call__(self, a, b, c):
        p = (a + b + c) / 2
        result = (p * (p - a) * (p - b) * (p - c)) ** 0.5
        if random() <= self.p:
            return result
        else:
            return uniform(result-self.v,
                                   result+self.v)


class MergeExperts:

    def __init__(self, mode, *experts):
        self.mode, self.experts = mode, experts

    def __call__(self, a, b, c):
        results= [exp(a, b, c) for exp in self.experts]
        if self.mode == "vote":
            c = Counter(results)
            return c.most_common(1)[0][0]
        elif self.mode == "mean":
            return sum(results) / len(results)

rvalues = [(uniform(0.7, 0.9), uniform(0.05, 0.2)) for _ in rang
e(20)]
experts = [FuzzyTriangleArea(p, v) for p, v in rvalues]
merger1 = MergeExperts("vote", *experts)
print(merger1(3, 4, 5))
merger2 = MergeExperts("mean", *experts)
print(merger2(3, 4, 5))
```

```
6.0
6.0073039634137375
```

The following example defines a class with which we can create abitrary polynomial functions:

```python
class Polynomial:

    def __init__(self, *coefficients):
        self.coefficients = coefficients[::-1]

    def __call__(self, x):
        res = 0
        for index, coeff in enumerate(self.coefficients):
            res += coeff * x** index
        return res

# a constant function
p1 = Polynomial(42)

# a straight Line
p2 = Polynomial(0.75, 2)

# a third degree Polynomial
p3 = Polynomial(1, -0.5, 0.75, 2)

for i in range(1, 10):
    print(i, p1(i), p2(i), p3(i))
```

```
1 42 2.75 3.25
2 42 3.5 9.5
3 42 4.25 26.75
4 42 5.0 61.0
5 42 5.75 118.25
6 42 6.5 204.5
7 42 7.25 325.75
8 42 8.0 488.0
9 42 8.75 697.25
```

You will find further interesting examples of the `__call__` function in our tutorial in the chapters Decorators and Memoization with Decorators. You may also consult our chapter on Polynomials.

INTRODUCTION

There aren't many good examples on inheritance available on the web. They are either extremely simple and artificial or they are way too complicated. We want to close the gap by providing an example which is on the one hand more realistic - but still not realistic - and on the other hand simple enough to see and understand the basic aspects of inheritance. In our previous chapter, we introduced inheritance formally.

To this purpose we define two base classes: One is an implementation of a clock and the other one of a calendar. Based on these two classes, we define a class CalendarClock, which inherits both from the class Calendar and from the class Clock.

THE CLOCK CLASS

```python
class Clock(object):

    def __init__(self,hours=0, minutes=0, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def set(self,hours, minutes, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def tick(self):
        """ Time will be advanced by one second """
        if self.__seconds == 59:
            self.__seconds = 0
            if (self.__minutes == 59):
                self.__minutes = 0
                self.__hours = 0 if self.__hours==23  else sel
f.__hours + 1
            else:
                self.__minutes += 1;
        else:
```

```
            self.__seconds += 1;

    def display(self):
        print("%d:%d:%d" % (self.__hours, self.__minutes, self.__s
econds))

    def __str__(self):
        return "%2d:%2d:%2d" % (self.__hours, self.__minutes, sel
f.__seconds)

x = Clock()
print(x)
for i in range(10000):
    x.tick()
print(x)
```

```
 0: 0: 0
 2:46:40
```

THE CALENDAR CLASS

```
class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    def __init__(self, day=1, month=1, year=1900):
        self.__day = day
        self.__month = month
        self.__year = year

    def leapyear(self,y):
        if y % 4:
            # not a leap year
            return 0;
        else:
            if y % 100:
                return 1;
            else:
                if y % 400:
                    return 0
                else:
                    return 1;

    def set(self, day, month, year):
        self.__day = day
        self.__month = month
```

```python
            self.__year = year


    def get():
        return (self, self.__day, self.__month, self.__year)
    def advance(self):
        months = Calendar.months
        max_days = months[self.__month-1]
        if self.__month == 2:
            max_days += self.leapyear(self.__year)
        if self.__day == max_days:
            self.__day = 1
            if (self.__month == 12):
                self.__month = 1
                self.__year += 1
            else:
                self.__month += 1
        else:
            self.__day += 1



    def __str__(self):
        return str(self.__day)+"/"+ str(self.__month)+ "/"+ str(sel
f.__year)

if __name__ == "__main__":
    x = Calendar()
    print(x)
    x.advance()
    print(x)
```

```
1/1/1900
2/1/1900
```

THE CALENDAR-CLOCK CLASS

```python
from clock1 import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):

    def __init__(self, day,month,year,hours=0, minutes=0,second
s=0):
        Calendar.__init__(self, day, month, year)
        Clock.__init__(self, hours, minutes, seconds)
```

```
    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)


if __name__ == "__main__":
    x = CalendarClock(24,12,57)
    print(x)
    for i in range(1000):
        x.tick()
    for i in range(1000):
        x.advance()
    print(x)
```

```
 0: 0: 0
 2:46:40
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-ced1ddf95fd9> in <module>
     13
     14 if __name__ == "__main__":
---> 15     x = CalendarClock(24,12,57)
     16     print(x)
     17     for i in range(1000):

<ipython-input-5-ced1ddf95fd9> in __init__(self, day, month, year, hours, m
inutes, seconds)
      5
      6     def __init__(self, day,month,year,hours=0, minutes=0,seconds=0)
----> 7         Calendar.__init__(self, day, month, year)
      8         Clock.__init__(self, hours, minutes, seconds)
      9

TypeError: __init__() takes from 1 to 2 positional arguments but 4 were giv
en
```

POLYNOMIALS
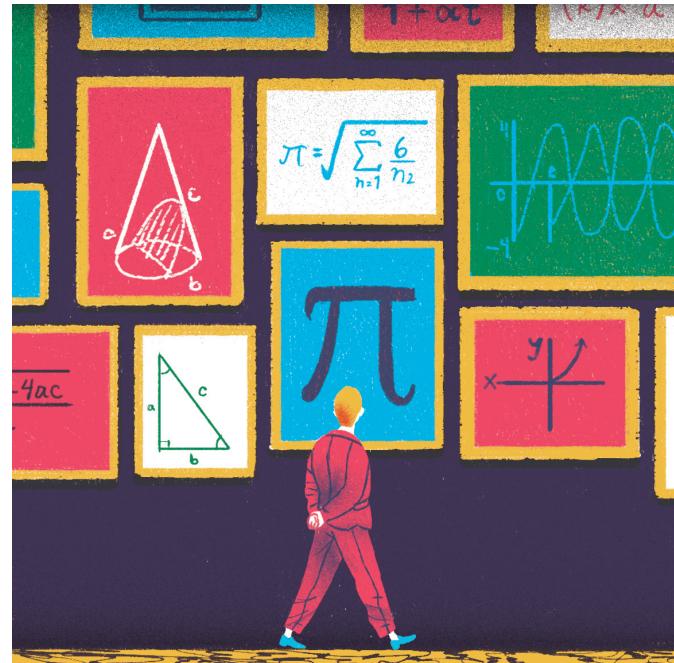
INTRODUCTION

If you have been to highschool, you will have encountered the terms **polynomial** and **polynomial function**. This chapter of our Python tutorial is completely on polynomials, i.e. we will define a class to define polynomials. The following is an example of a polynomial with the degree 4:

$$p(x) = x^4 - 4 \cdot x^2 + 3 \cdot x$$

You will find out that there are lots of similarities to integers. We will define various arithmetic operations for polynomials in our class, like addition, subtraction, multiplication and division. Our polynomial class will also provide means to calculate the derivation and the integral of polynomials. We will not miss out on plotting polynomials.

There is a lot of beaty in polynomials and above all in how they can be implemented as a Python class. We like to say thanks to Drew Shanon who granted us permission to use his great picture, treating math as art!

SHORT MATHEMATICAL INTRODUCTION

We will only deal with polynomial in a single indeterminate (also called variable) x. A general form of a polynomial in a single indeterminate looks like this:

$$a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

where $a_0, a_1, \ldots a_n$ are the constants - non-negative integers - and x is the indeterminate or variable. The term "indeterminate" means that x represents no particular value, but any value may be substituted for it.

This expression is usually written with the summation operator:

$$\sum_{k=0}^{n} a_k \cdot x^k = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

A polynomial function is a function that can be defined by evaluating a polynomial. A function f of one argument can be defined as:

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

POLYNOMIAL FUNCTIONS WITH PYTHON

It's easy to implement polynomial functions in Python. As an example we define the polynomial function given in the introduction of this chapter, i.e. $p(x) = x^4 - 4 \cdot x^2 + 3 \cdot x$

The Python code for this polynomial function looks like this:

```python
def p(x):
    return x**4 - 4*x**2 + 3*x
```

We can call this function like any other function:
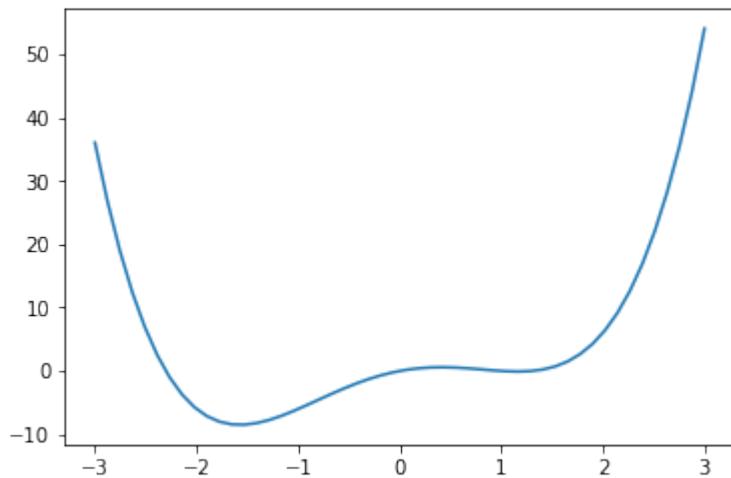
```python
for x in [-1, 0, 2, 3.4]:
    print(x, p(x))
```

```
-1 -6
0 0
2 6
3.4 97.59359999999998
```

```python
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-3, 3, 50, endpoint=True)
F = p(X)
plt.plot(X,F)

plt.show()
```

POLYNOMIAL CLASS

We will define now a class for polynomial functions. We will build on an idea which we have developed in the chapter on decorators of our Python tutorial. We introduced polynomial factories.

A polynomial is uniquely determined by its coefficients. This means, an instance of our polynomial class needs a list or tuple to define the coefficients.

```python
class Polynomial:

    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned i
nto a list

    def __repr__(self):
        """
        method to return the canonical string representation
        of a polynomial.
        """
        return "Polynomial" + str(tuple(self.coefficients))
```

We can the instantiate the polynomial of our previous example polynomial function like this:

```python
p = Polynomial(1, 0, -4, 3, 0)
print(p)

p2 = eval(repr(p))
```

```
print(p2)
```

```
Polynomial(1, 0, -4, 3, 0)
Polynomial(1, 0, -4, 3, 0)
```

Having defined the canonical string representation of a Polynomial with the **repr** function, we also want to define an output which could be used to create a version which is more friendly for people. We write a LaTex represention which can be used to print the function in a beatiful way:

```python
class Polynomial:

    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned into a list

    def __repr__(self):
        """
        method to return the canonical string representation
        of a polynomial.
        """
        return "Polynomial" + str(tuple(self.coefficients))

    def __str__(self):

        def x_expr(degree):
            if degree == 0:
                res = ""
            elif degree == 1:
                res = "x"
            else:
                res = "x^"+str(degree)
            return res

        degree = len(self.coefficients) - 1
        res = ""

        for i in range(0, degree+1):
            coeff = self.coefficients[i]
            # nothing has to be done if coeff is 0:
            if abs(coeff) == 1 and i < degree:
```

```
                    # 1 in front of x shouldn't occur, e.g. x instead
of 1x
                    # but we need the plus or minus sign:
                    res += f"{'+' if coeff>0 else '-'}{x_expr(degre
e-i)}"
            elif coeff != 0:
                    res += f"{coeff:+g}{x_expr(degree-i)}"

        return res.lstrip('+')    # removing leading '+'
```

```
polys = [Polynomial(1, 0, -4, 3, 0),
         Polynomial(2, 0),
         Polynomial(4, 1, -1),
         Polynomial(3, 0, -5, 2, 7),
         Polynomial(-42)]

# output suitable for usage in LaTeX:
for count, poly in enumerate(polys):
    print(f"$p_{count} = {str(poly)}$")
```

```
$p_0 = x^4-4x^2+3x$
$p_1 = 2x$
$p_2 = 4x^2+x-1$
$p_3 = 3x^4-5x^2+2x+7$
$p_4 = -42$
```

If we use this in LaTeX it will look like this:

$$p_0 = x^4 - 4x^2 + 3x$$

$$p_1 = 2x$$

$$p_2 = 4x^2 + x - 1$$

$$p_3 = 3x^4 - 5x^2 + 2x + 7$$

$$p_4 = -42$$

So far, we have defined **polynomials**, but what we actually need are **polynomial functions**. For this purpose, we turn instances of the Polynomial class into **callables** by defining the **call** method.

```
class Polynomial:

    def __init__(self, *coefficients):
```

```
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned i
nto a list

    # The __repr__ and __str__ method can be included here,
    # but is not necessary for the immediately following code

    def __call__(self, x):
        res = 0
        for index, coeff in enumerate(self.coefficients[::-1]):
            res += coeff * x** index
        return res
```

It is possible now to call an instance of our class like a function. We call it with an argument and the instance,
- which is a callable, - behaves like a polynomial function:

```
p = Polynomial(3, 0, -5, 2, 1)
print(p)

for x in range(-3, 3):
    print(x, p(x))
```

```
<__main__.Polynomial object at 0x7fd38cd9d250>
-3 193
-2 25
-1 -3
0 1
1 1
2 33
```

Just for fun, let us plot the previously defined function:
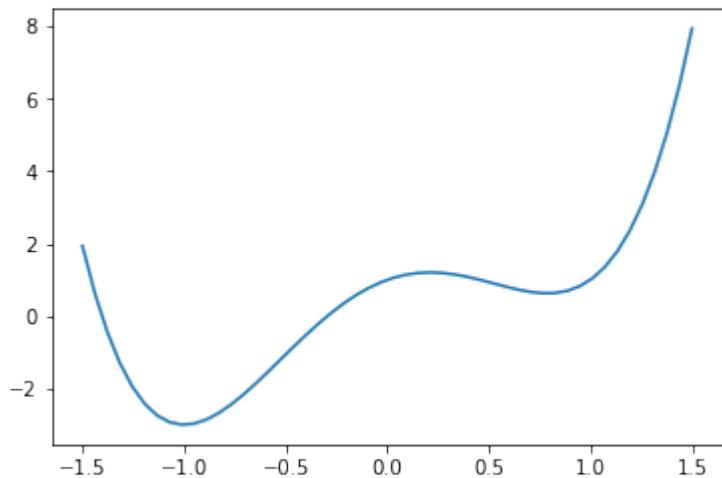
```
import matplotlib.pyplot as plt

X = np.linspace(-1.5, 1.5, 50, endpoint=True)
F = p(X)
plt.plot(X, F)

plt.show()
```

Before we further refine our class, let us use a numerically more efficient variant for the calculation of polynomials. We will use this algorithm in our `__call__` method. We introduced this variant in our chapter on decorators.

Every polynomial

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

can be also written in the form

$$f(x) = (a_n x + x_{n-1})x + \cdot + a_1)x + a_0$$

To emperically see that they are equivalent, we rewrite our class definition, but call it `Polynomial2` so that we can use both versions:

```python
class Polynomial2:

    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned into a list

        # The __repr__ and __str__ method can be included here,
        # but is not necessary for the immediately following code

    def __call__(self, x):
        res = 0
        for coeff in self.coefficients:
            res = res * x + coeff
```

```
        return res
```

We will test both classes by comparing the results:

```
p1 = Polynomial(-4, 3, 0)
p2 = Polynomial2(-4, 3, 0)
res = all((p1(x)==p2(x) for x in range(-10, 10)))
print(res)
```

```
True
```

It is possible to define addition and subtractions for polynomials. All we have to do is to add or subtract the coefficients with the same exponents from both polynomials.

If we have polynomial functions $f(x) = \sum_{k=0}^{n} a_k \cdot x^k$ and $g(x) = \sum_{k=0}^{n} b_k \cdot x^k$, the addition is defined as

$$(f+g)(x) = \sum_{k=0}^{n} (a_k + b_k) \cdot x^k$$

and correspondingly subtraction is defined as

$$(f-g)(x) = \sum_{k=0}^{n} (a_k - b_k) \cdot x^k$$

Before we can add the methods __add__ and __sub__, which are necessary for addition and subtraction, we add a generatur zip_longest(). It works similar to zip for two parameters, but it does not stop if one of the iterators is exhausted but uses "fillvalue" instead.

```
def zip_longest(iter1, iter2, fillvalue=None):

    for i in range(max(len(iter1), len(iter2))):
        if i >= len(iter1):
            yield (fillvalue, iter2[i])
        elif i >= len(iter2):
            yield (iter1[i], fillvalue)
        else:
            yield (iter1[i], iter2[i])
        i += 1

p1 = (2,)
p2 = (-1, 4, 5)
for x in zip_longest(p1, p2, fillvalue=0):
    print(x)
```

```
(2, -1)
(0, 4)
(0, 5)
```

We will add this generator to our class as a static method. We are now capable of adding the __add__ and __sub__ methods as well. We will need the generator `zip_longest` from itertools. `zip_longest` takes and arbitrary number of iterables and a keyword parameter `fillvalue`. It creates an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

```python
import numpy as np
import matplotlib.pyplot as plt
from itertools import zip_longest

class Polynomial:

    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned into a list

    def __repr__(self):
        """
        method to return the canonical string representation
        of a polynomial.

        """
        return "Polynomial" + str(self.coefficients)

    def __call__(self, x):
        res = 0
        for coeff in self.coefficients:
            res = res * x + coeff
        return res

    def degree(self):
        return len(self.coefficients)

    def __add__(self, other):
        c1 = self.coefficients[::-1]
        c2 = other.coefficients[::-1]
        res = [sum(t) for t in zip_longest(c1, c2, fillvalue=0)]
        return Polynomial(*res[::-1])
```

```python
    def __sub__(self, other):
        c1 = self.coefficients[::-1]
        c2 = other.coefficients[::-1]

        res = [t1-t2 for t1, t2 in zip_longest(c1, c2, fillvalue=0)]
        return Polynomial(*res[::-1])



p1 = Polynomial(4, 0, -4, 3, 0)
p2 = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)

p_sum = p1 + p2
p_diff = p1 - p2

X = np.linspace(-3, 3, 50, endpoint=True)
F1 = p1(X)
F2 = p2(X)
F_sum = p_sum(X)
F_diff = p_diff(X)
plt.plot(X, F1, label="F1")
plt.plot(X, F2, label="F2")
plt.plot(X, F_sum, label="F_sum")
plt.plot(X, F_diff, label="F_diff")

plt.legend()
plt.show()
```
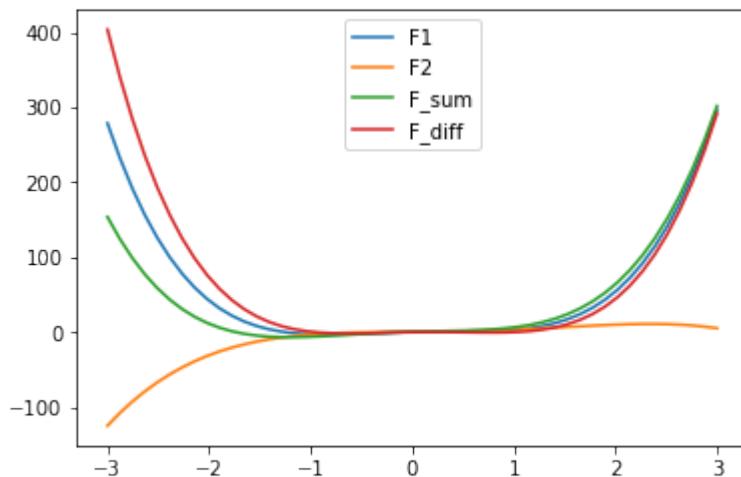
It is incredibly easy to add differentiation to our class. Mathematically, it is defined as

$$f'(x) = \sum_{k=0}^{n} k \cdot a_k \cdot x^{k-1}$$

if

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

This can be easily implemented in our method 'derivative':

```python
import numpy as np
import matplotlib.pyplot as plt


class Polynomial:


    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ...a_1, a_0
        """
        self.coefficients = list(coefficients) # tuple is turned into a list


    def __repr__(self):
        """
        method to return the canonical string representation
        of a polynomial.

        """
        return "Polynomial" + str(self.coefficients)


    def __call__(self, x):
        res = 0
        for coeff in self.coefficients:
            res = res * x + coeff
        return res


    def degree(self):
```

```python
        return len(self.coefficients)


    def __add__(self, other):
        c1 = self.coefficients[::-1]
        c2 = other.coefficients[::-1]
        res = [sum(t) for t in zip_longest(c1, c2, fillvalue=0)]
        return Polynomial(*res)


    def __sub__(self, other):
        c1 = self.coefficients[::-1]
        c2 = other.coefficients[::-1]

        res = [t1-t2 for t1, t2 in zip_longest(c1, c2, fillvalu
e=0)]
        return Polynomial(*res)


    def derivative(self):
        derived_coeffs = []
        exponent = len(self.coefficients) - 1
        for i in range(len(self.coefficients)-1):
            derived_coeffs.append(self.coefficients[i] * exponent)
            exponent -= 1
        return Polynomial(*derived_coeffs)


    def __str__(self):

        def x_expr(degree):
            if degree == 0:
                res = ""
            elif degree == 1:
                res = "x"
            else:
                res = "x^"+str(degree)
            return res

        degree = len(self.coefficients) - 1
        res = ""

        for i in range(0, degree+1):
            coeff = self.coefficients[i]
            # nothing has to be done if coeff is 0:
```

```python
            if abs(coeff) == 1 and i < degree:
                # 1 in front of x shouldn't occur, e.g. x instead
of 1x
                # but we need the plus or minus sign:
                res += f"{'+' if coeff>0 else '-'}{x_expr(degre
e-i)}"
            elif coeff != 0:
                res += f"{coeff:+g}{x_expr(degree-i)}"

        return res.lstrip('+')    # removing leading '+'


p = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)

p_der = p.derivative()

X = np.linspace(-2, 3, 50, endpoint=True)

F = p(X)
F_derivative = p_der(X)
plt.plot(X, F, label="F")
plt.plot(X, F_derivative, label="F_der")

plt.legend()
plt.show()
```


Playing around a little bit:

```python
p = Polynomial(1, 2, -3, 4, -55)
```

```
p2 = Polynomial(1, 2, 3)

p_der = p.derivative()

print(p)
print(p_der)
print(p2)

p3 = p + p2

print(p3)
```

```
x^4+2x^3-3x^2+4x-55
4x^3+6x^2-6x+4
x^2+2x+3
-52x^4+6x^3-2x^2+2x+1
```

SLOTS

AVOIDING DYNAMICALLY CREATED ATTRIBUTES

The attributes of objects are stored in a dictionary `__dict__`. Like any other dictionary, a dictionary used for attribute storage doesn't have a fixed number of elements. In other words, you can add elements to dictionaries after they are defined, as we have seen in our chapter on dictionaries. This is the reason, why you can dynamically add attributes to objects of classes that we have created so far:

```python
class A(object):
    pass

a = A()
a.x = 66
a.y = "dynamically created attribut
e"
```

The dictionary containing the attributes of "a" can be accessed like this:
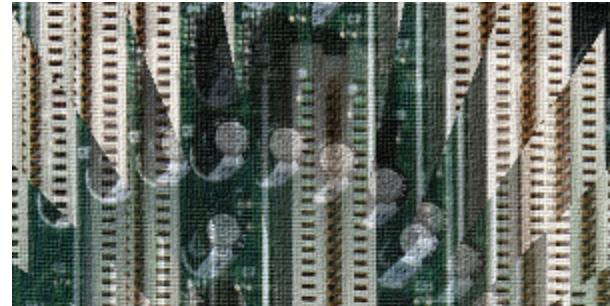
```python
a.__dict__
```

Output:
```
{'x': 66, 'y': 'dynamically cr
   eated attribute'}
```

You might have wondered that you can dynamically add attributes to the classes, we have defined so far, but that you can't do this with built-in classes like 'int', or 'list':

```python
x = 42
x.a = "not possible to do it"
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-8c5f7956a976> in <module>
      1 x = 42
----> 2 x.a = "not possible to do it"

AttributeError: 'int' object has no attribute 'a'
```

```
lst = [34, 999, 1001]
lst.a = "forget it"
```

```
---------------------------------------------------------------------
AttributeError                          Traceback (most recent call last)
<ipython-input-4-df06616479b6> in <module>
      1 lst = [34, 999, 1001]
----> 2 lst.a = "forget it"

AttributeError: 'list' object has no attribute 'a'
```

If we generated a class in which usually only a few instances are needed in a program, - such as the Function class, - the advantages outweigh the disadvantages. The additional storage space for the dictionary brings us significant advantages for the design of our software. However, as soon as a high number of instances of a class must be generated in a program, the cost-benefit ratio can quickly reverse. The additionally required storage space can adversely affect or even prevent the execution of the program.

Python's slots are a nice way to work around this space consumption problem. Instead of having a dynamic `dict` dictionary that allows adding attributes to objects dynamically, slots provide a static structure which prohibits additions after the creation of an instance.

When we design a class, we can use slots to prevent the dynamic creation of attributes. To define slots, you have to define a list with the name `__slots__`. The list has to contain all the attributes, you want to use. Anything not in this list cannot be used as an attribute. We demonstrate this in the following class, in which the `__slots__` list contains only the name for an attribute `val`:

```
class S(object):

    __slots__ = ['val']

    def __init__(self, v):
        self.val = v


x = S(42)
print(x.val)

x.new = "not possible"
```

```
42
```

```
AttributeError                                          Traceback (most recent call last)
<ipython-input-5-58aeffbcf9f0> in <module>
     10 print(x.val)
     11
---> 12 x.new = "not possible"

AttributeError: 'S' object has no attribute 'new'
```

If we start this program, we can see, that it is not possible to create dynamically a new attribute. We fail to create an attribute "new".

We mentioned in the beginning that slots are preventing a waste of space with objects. Since Python 3.3 this advantage is not as impressive any more. With Python 3.3 Key-Sharing Dictionaries are used for the storage of objects. The attributes of the instances are capable of sharing part of their internal storage between each other, i.e. the part which stores the keys and their corresponding hashes. This helps reducing the memory consumption of programs, which create many instances of non-builtin types.

CLASSES AND CLASS CREATION

BEHIND THE SCENES: RELATIONSHIP BETWEEN CLASS AND TYPE

In this chapter of our tutorial, we will provide you with a deeper insight into the magic happening behind the scenes, when we are defining a class or creating an instance of a class. You may ask yourself: "Do I really have to learn theses additional details on object oriented programming in Python?" Most probably not, or you belong to the few people who design classes at a very advanced level.

First, we will concentrate on the relationship between type and class. While defining classes so far, you may have asked yourself, what is happening "behind the lines". We have already seen, that applying "type" to an object returns the class of which the object is an instance of:

```
x = [4, 5, 9]
y = "Hello"
print(type(x), type(y))
```

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

If you apply type on the name of a class itself, you get the class "type" returned.

```
print(type(list), type(str))
```

```
<class 'type'> <class 'type'>
```

This is similar to applying type on type(x) and type(y):

```
x = [4, 5, 9]
y = "Hello"
print(type(x), type(y))
print(type(type(x)), type(type(y)))
```

```
<class 'list'> <class 'str'>
<class 'type'> <class 'type'>
```

A user-defined class (or the class "object") is an instance of the class "type". So, we can see, that classes are created from type. In Python3 there is no difference between "classes" and "types". They are in most cases used as synonyms.

The fact that classes are instances of a class "type" allows us to program metaclasses. We can create classes, which inherit from the class "type". So, a metaclass is a subclass of the class "type".

Instead of only one argument, type can be called with three parameters:

type(classname, superclasses, attributes_dict)

If type is called with three arguments, it will return a new type object. This provides us with a dynamic form of the class statement.

- "classname" is a string defining the class name and becomes the name attribute;
- "superclasses" is a list or tuple with the superclasses of our class. This list or tuple will become the bases attribute;
- the attributes_dict is a dictionary, functioning as the namespace of our class. It contains the definitions for the class body and it becomes the dict attribute.

Let's have a look at a simple class definition:

```
class A:
    pass

x = A()
print(type(x))
```

```
<class '__main__.A'>
```

We can use "type" for the previous class defintion as well:

```
A = type("A", (), {})
x = A()
print(type(x))
```

```
<class '__main__.A'>
```

Generally speaking, this means, that we can define a class A with

```
    type(classname, superclasses, attributedict)
```

When we call "type", the call method of type is called. The call method runs two other methods: new and init:

```
    type.__new__(typeclass, classname, superclasses, attributedict)

    type.__init__(cls, classname, superclasses, attributedict)
```

The new method creates and returns the new class object, and after this, the init method initializes the newly created object.

```python
class Robot:

    counter = 0

    def __init__(self, name):
        self.name = name

    def sayHello(self):
        return "Hi, I am " + self.name


def Rob_init(self, name):
    self.name = name

Robot2 = type("Robot2",
              (),
              {"counter":0,
               "__init__": Rob_init,
               "sayHello": lambda self: "Hi, I am " + self.name})

x = Robot2("Marvin")
print(x.name)
print(x.sayHello())

y = Robot("Marvin")
print(y.name)
print(y.sayHello())

print(x.__dict__)
print(y.__dict__)
```
```
Marvin
Hi, I am Marvin
Marvin
Hi, I am Marvin
{'name': 'Marvin'}
{'name': 'Marvin'}
```

The class definitions for Robot and Robot2 are syntactically completely different, but they implement logically the same class.

What Python actually does in the first example, i.e. the "usual way" of defining classes, is the following: Python processes the complete class statement from class Robot to collect the methods and attributes of Robot to add them to the attributes_dict of the type call. So, Python will call type in a similar or the same way as we did in Robot2.

MOTIVATION FOR METACLASSES

In this chapter of our tutorial we want to provide some incentives or motivation for the use of metaclasses. To demonstrate some design problems, which can be solved by metaclasses, we will introduce and design a bunch of philosopher classes. Each philosopher class (Philosopher1, Philosopher2, and so on) need the same "set" of methods (in our example just one, i.e. "the_answer") as the basics for his or her pondering and brooding. A stupid way to implement the classes would be having the same code in every philospher class:

```python
class Philosopher1(object):

    def the_answer(self, *args):
        return 42

class Philosopher2(object):

    def the_answer(self, *args):
        return 42

class Philosopher3(object):

    def the_answer(self, *args):
        return 42
```

```
plato = Philosopher1()
print (plato.the_answer())

kant = Philosopher2()
# let's see what Kant has to say :-)
print (kant.the_answer())
```

```
42
42
```

We can see that we have multiple copies of the method "the_answer". This is error prone and tedious to maintain, of course.

From what we know so far, the easiest way to accomplish our goal without creating redundant code would be designing a base, which contains "the_answer" as a method. Now each Philosopher class inherits from this base class:

```python
class Answers(object):

    def the_answer(self, *args):
        return 42


class Philosopher1(Answers):
    pass

class Philosopher2(Answers):
    pass

class Philosopher3(Answers):
    pass

plato = Philosopher1()
print plato.the_answer()

kant = Philosopher2()
# let's see what Kant has to say :-)
print kant.the_answer()
```

```
42
42
```

The way we have designed our classes, each Philosopher class will always have a method "the_answer". Let's assume, we don't know a priori if we want or need this method. Let's assume that the decision, if the classes have to be augmented, can only be made at runtime. This decision might depend on configuration files, user

input or some calculations.

```python
# the following variable would be set as the result of a runtime c
alculation:
x = raw_input("Do you need 'the answer'? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

class Philosopher1(object):
    pass
if required:
    Philosopher1.the_answer = the_answer

class Philosopher2(object):
    pass
if required:
    Philosopher2.the_answer = the_answer

class Philosopher3(object):
    pass
if required:
    Philosopher3.the_answer = the_answer


plato = Philosopher1()
kant = Philosopher2()

# let's see what Plato and Kant have to say :-)
if required:
    print kant.the_answer()
    print plato.the_answer()
else:
    print "The silence of the philosphers"
```

```
Do you need 'the answer'? (y/n): y
42
42
```

Even though this is another solution to our problem, there are still some serious drawbacks. It's error-prone,

because we have to add the same code to every class and it seems likely that we might forget it. Besides this it's getting hardly manageable and maybe even confusing, if there are many methods we want to add.

We can improve our approach by defining a manager function and avoiding redundant code this way. The manager function will be used to augment the classes conditionally.

```python
# the following variable would be set as the result of a runtime c
alculation:
x = raw_input("Do you need 'the answer'? (y/n): ")
if x=="y":
    required = True
else:
    required = False


def the_answer(self, *args):
        return 42

# manager function
def augment_answer(cls):
    if required:
        cls.the_answer = the_answer


class Philosopher1(object):
    pass
augment_answer(Philosopher1)

class Philosopher2(object):
    pass
augment_answer(Philosopher2)

class Philosopher3(object):
    pass
augment_answer(Philosopher3)


plato = Philosopher1()
kant = Philosopher2()

# let's see what Plato and Kant have to say :-)
if required:
    print kant.the_answer()
    print plato.the_answer()
```

```
else:
    print "The silence of the philosphers"

Do you need 'the answer'? (y/n): y
42
42
```

This is again useful to solve our problem, but we, i.e. the class designers, must be careful not to forget to call the manager function "augment_answer". The code should be executed automatically. We need a way to make sure that "some" code might be executed automatically after the end of a class definition.

```python
# the following variable would be set as the result of a runtime c
alculation:
x = raw_input("Do you need 'the answer'? (y/n): ")
if x=="y":
    required = True
else:
    required = False
def the_answer(self, *args):
        return 42

def augment_answer(cls):
    if required:
        cls.the_answer = the_answer
    # we have to return the class now:
    return cls

@augment_answer
class Philosopher1(object):
    pass

@augment_answer
class Philosopher2(object):
    pass

@augment_answer
class Philosopher3(object):
    pass

plato = Philosopher1()
kant = Philosopher2()

# let's see what Plato and Kant have to say :-)
if required:
    print kant.the_answer()
```
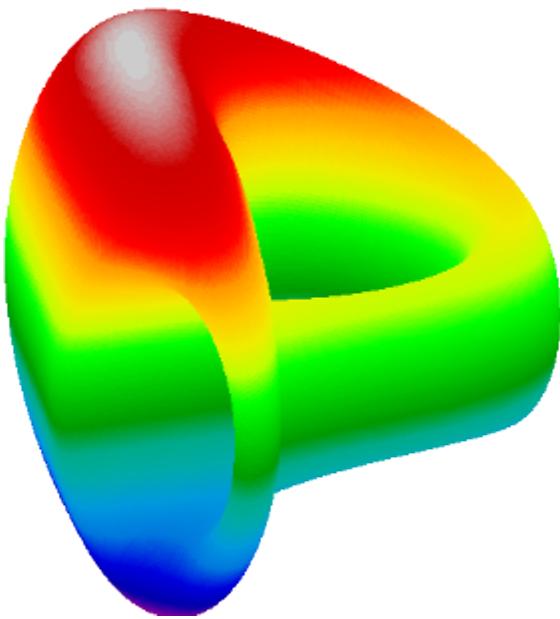
```
    print plato.the_answer()
else:
    print "The silence of the philosphers"
```
```
Do you need 'the answer'? (y/n): y
42
42
```

Metaclasses can also be used for this purpose as we will learn in the next chapter.

METACLASSES

A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.

Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way they implement them. Python is supporting them.

Some programmers see metaclasses in Python as "solutions waiting or looking for a problem".

There are numerous use cases for metaclasses. Just to name a few:

- logging and profiling
- interface checking
- registering classes at creation time
- automatically adding new methods
- automatic property creation
- proxies
- automatic resource locking/synchronization.

DEFINING METACLASSES

Principially, metaclasses are defined like any other Python class, but they are classes that inherit from "type". Another difference is, that a metaclass is called automatically, when the class statement using a metaclass

ends. In other words: If no "metaclass" keyword is passed after the base classes (there may be no base classes either) of the class header, type() (i.e. __call__ of type) will be called. If a metaclass keyword is used, on the other hand, the class assigned to it will be called instead of type.

Now we create a very simple metaclass. It's good for nothing, except that it will print the content of its arguments in the __new__ method and returns the results of the `type.__new__` call:

```python
class LittleMeta(type):
    def __new__(cls, clsname, superclasses, attributedict):
        print("clsname: ", clsname)
        print("superclasses: ", superclasses)
        print("attributedict: ", attributedict)
        return type.__new__(cls, clsname, superclasses, attributedict)
```

We will use the metaclass "LittleMeta" in the following example:

```python
class S:
    pass


class A(S, metaclass=LittleMeta):
    pass


a = A()
```

```
clsname:  A
superclasses:  (<class '__main__.S'>,)
attributedict:  {'__module__': '__main__', '__qualname__': 'A'}
```

We can see `LittleMeta.__new__` has been called and not `type.__new__`.

Resuming our thread from the last chapter: We define a metaclass `EssentialAnswers` which is capable of automatically including our `augment_answer` method:

```python
x = input("Do you need the answer? (y/n): ")
if x.lower() == "y":
    required = True
else:
    required = False


def the_answer(self, *args):
```

```
            return 42


class EssentialAnswers(type):

    def __init__(cls, clsname, superclasses, attributedict):
        if required:
            cls.the_answer = the_answer


class Philosopher1(metaclass=EssentialAnswers):
    pass


class Philosopher2(metaclass=EssentialAnswers):
    pass


class Philosopher3(metaclass=EssentialAnswers):
    pass


plato = Philosopher1()
print(plato.the_answer())


kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())
```
```
Do you need the answer? (y/n): y
42
42
```

We have learned in our chapter "Type and Class Relationship" that after the class definition has been processed, Python calls

```
    type(classname, superclasses, attributes_dict)
```

This is not the case, if a metaclass has been declared in the header. That is what we have done in our previous example. Our classes Philosopher1, Philosopher2 and Philosopher3 have been hooked to the metaclass EssentialAnswers. That's why EssentialAnswer will be called instead of type:

```
    EssentialAnswer(classname, superclasses, attributes_dict)
```

To be precise, the arguments of the calls will be set the the following values:

```
EssentialAnswer('Philopsopher1',
                (),
                {'__module__': '__main__', '__qualname__': 'Philoso
pher1'})
```

The other philosopher classes are treated in an analogue way.

CREATING SINGLETONS USING METACLASSES

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. It is used in cases where exactly one object is needed. The concept can be generalized to restrict the instantiation to a certain or fixed number of objects. The term stems from mathematics, where a singleton, - also called a unit set -, is used for sets with exactly one element.

```python
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleto
n,                               cls).__call__(*args, **kwa
rgs)
        return cls._instances[cls]


class SingletonClass(metaclass=Singleton):
    pass


class RegularClass():
    pass


x = SingletonClass()
y = SingletonClass()
print(x == y)


x = RegularClass()
y = RegularClass()
print(x == y)
```

```
True
False
```

CREATING SINGLETONS USING METACLASSES

Alternatively, we can create Singleton classes by inheriting from a Singleton class, which can be defined like this:

```python
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance


class SingletonClass(Singleton):
    pass

class RegularClass():
    pass


x = SingletonClass()
y = SingletonClass()
print(x == y)


x = RegularClass()
y = RegularClass()
print(x == y)
```

```
True
False
```

COUNT METHOD CALLS USING A METACLASS

INTRODUCTION

After you have hopefully gone through our chapter [Introduction into Metaclasses](#) you may have asked yourself about the possible use cases for metaclasses. There are some interesting use cases and it's not - like some say - a solution waiting for a problem. We have mentioned already some examples.

In this chapter of our tutorial on Python, we want to elaborate an example metaclass, which will decorate the methods of the subclass. The decorated function returned by the decorator makes it possible to count the number of times each method of the subclass has been called.

This is usually one of the tasks, we expect from a profiler. So we can use this metaclass for simple profiling purposes. Of course, it will be easy to extend our metaclass for further profiling tasks.

PRELIMINARY REMARKS

Before we actually dive into the problem, we want to remind how we can access the attributes of a class. We will demonstrate this with the list class. We can get the list of all the non private attributes of a class - in our example the random class - with the following construct.

```python
import random
cls = "random" # name of the class as a string
```

```
all_attributes = [x for x in dir(eval(cls)) if not x.startswit
h("__") ]
print(all_attributes)
```

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICC
ONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodTyp
e', '_Sequence', '_Set', '_acos', '_ceil', '_cos', '_e', '_exp',
'_inst', '_log', '_pi', '_random', '_sha512', '_sin', '_sqrt', '_t
est', '_test_generator', '_urandom', '_warn', 'betavariate', 'choi
ce', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getst
ate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randin
t', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffl
e', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

Now, we are filtering the callable attributes, i.e. the public methods of the class.

```
methods = [x for x in dir(eval(cls)) if not x.startswith("__")
                            and callable(eval(cls + "." + x))]
print(methods)
```

```
['Random', 'SystemRandom', '_BuiltinMethodType', '_MethodType',
'_Sequence', '_Set', '_acos', '_ceil', '_cos', '_exp', '_log', '_s
ha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom',
'_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate',
'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvaria
te', 'paretovariate', 'randint', 'random', 'randrange', 'sample',
'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesv
ariate', 'weibullvariate']
```

Getting the non callable attributes of the class can be easily achieved by negating callable, i.e. adding "not":

```
non_callable_attributes = [x for x in dir(eval(cls)) if not x.star
tswith("__")
                            and not callable(eval(cls + "." +
x))]
print(non_callable_attributes)
```

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'SG_MAGICCONST', 'TW
OPI', '_e', '_inst', '_pi', '_random']
```

In normal Python programming it is neither recommended nor necessary to apply methods in the following way, but it is possible:

```
lst = [3,4]
list.__dict__["append"](lst, 42)
```

```
lst
```
Output: `[3, 4, 42]`

Please note the remark from the Python documentation:

"Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class."

A DECORATOR FOR COUNTING FUNCTION CALLS

Finally, we will begin designing the metaclass, which we have mentioned as our target in the beginning of this chapter. It will decorate all the methods of its subclass with a decorator, which counts the number of calls. We have defined such a decorator in our chapter Memoization and Decorators:

```python
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0
    helper.__name__ = func.__name__

    return helper
```

We can use it in the usual way:

```python
@call_counter
def f():
    pass

print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

```
0
10
```

It would be better if you add the alternative notation for decorating function. We will need this in our final metaclass:

```python
def f():
    pass

f = call_counter(f)
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

```
0
10
```

THE "COUNT CALLS" METACLASS

Now we have all the necessary "ingredients" together to write our metaclass. We will include our call_counter decorator as a staticmethod:

```python
class FuncCallCounter(type):
    """ A Metaclass which decorates all the methods of the
        subclass using call_counter as the decorator
    """

    @staticmethod
    def call_counter(func):
        """ Decorator for counting the number of function
            or method calls to the function or method func
        """
        def helper(*args, **kwargs):
            helper.calls += 1
            return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__ = func.__name__

        return helper
```

```python
    def __new__(cls, clsname, superclasses, attributedict):
        """ Every method gets decorated with the decorator call_co
unter,
            which will do the actual call counting
        """
        for attr in attributedict:
            if callable(attributedict[attr]) and not attr.startswi
th("__"):
                attributedict[attr] = cls.call_counter(attributedi
ct[attr])

        return type.__new__(cls, clsname, superclasses, attributed
ict)


class A(metaclass=FuncCallCounter):

    def foo(self):
        pass

    def bar(self):
        pass

if __name__ == "__main__":
    x = A()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    x.bar()
    print(x.foo.calls, x.bar.calls)
```

```
0 0
1 0
2 1
```

ABSTRACT CLASSES

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods.

You can see this in the following examples:

```python
class AbstractClass:

    def do_something(sel
f):
        pass



class B(AbstractClass):
    pass

a = AbstractClass()
b = B()
```

If we start this program, we see that this is not an abstract class, because:

- we can instantiate an instance from
- we are not required to implement do_something in the class defintition of B

Our example implemented a case of simple inheritance which has nothing to do with an abstract class. In fact, Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - for obvious reasons - abc.

The following Python code uses the abc module and defines an abstract base class:

```python
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
```

```
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

We will define now a subclass using the previously defined abstract class. You will notice that we haven't implemented the do_something method, even though we are required to implement it, because this method is decorated as an abstract method with the decorator "abstractmethod". We get an exception that DoAdd42 can't be instantiated:

```
class DoAdd42(AbstractClassExample):
    pass

x = DoAdd42(4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-2bcc42ab0b46> in <module>
      2     pass
      3
----> 4 x = DoAdd42(4)

TypeError: Can't instantiate abstract class DoAdd42 with abstract methods d
o_something
```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```
class DoAdd42(AbstractClassExample):

    def do_something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):

    def do_something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)

print(x.do_something())
print(y.do_something())
```

```
52
420
```

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

You may think that abstract methods can't be implemented in the abstract base class. This impression is wrong: An abstract method can have an implementation in the abstract class! Even if they are implemented, designers of subclasses will be forced to override the implementation. Like in other cases of "normal" inheritance, the abstract method can be invoked with super() call mechanism. This enables providing some basic functionality in the abstract method, which can be enriched by the subclass implementation.

```python
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):

    def do_something(self):
        super().do_something()
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

```
Some implementation!
The enrichment from AnotherSubclass
```