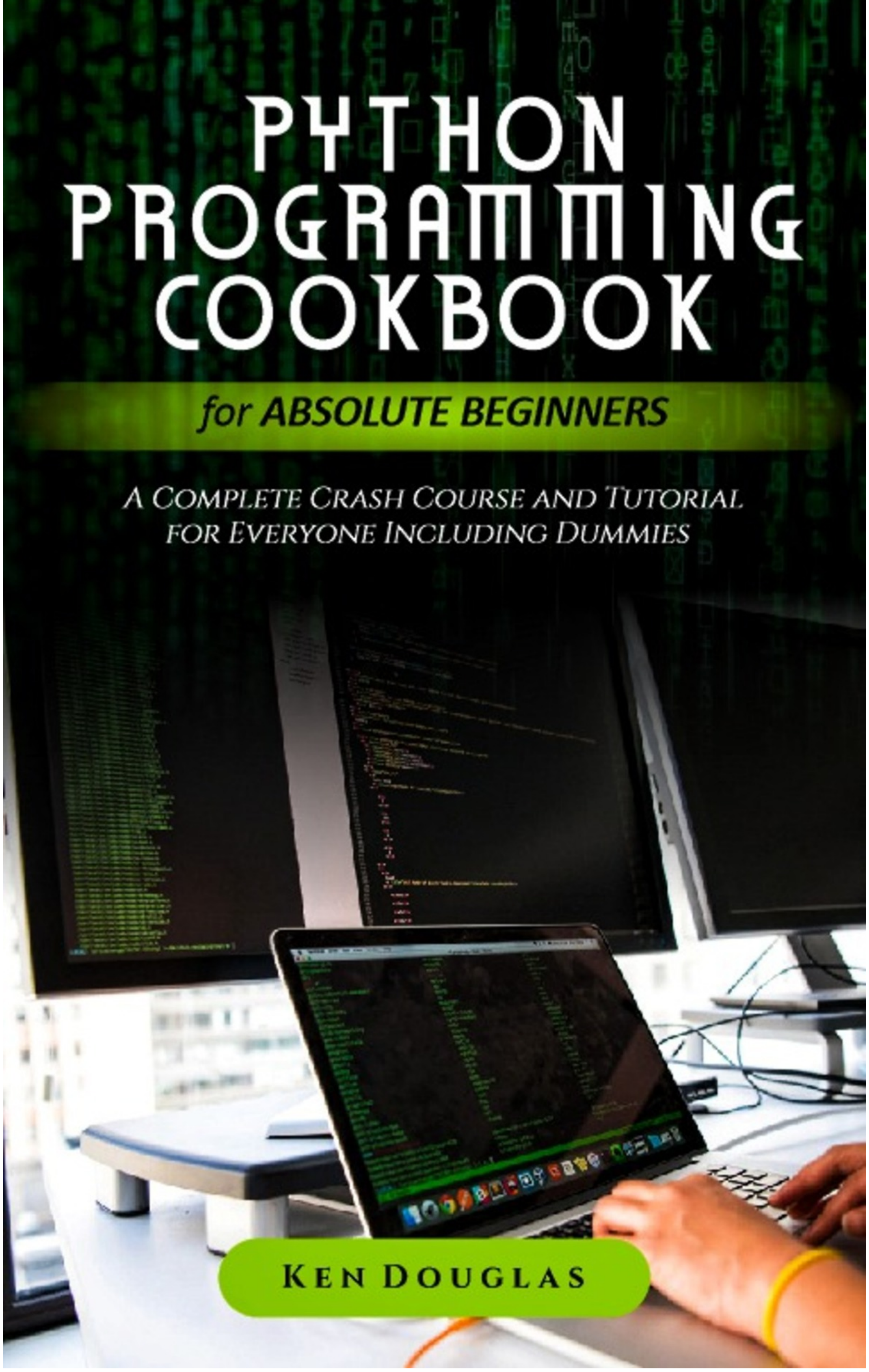
PYTHON PROGRAMMING COOKBOOK

for ABSOLUTE BEGINNERS

A Complete Crash Course and Tutorial for Everyone Including Dummies

KEN DOUGLAS


PYTHON PROGRAMMING COOKBOOK
for ABSOLUTE BEGINNERS
A COMPLETE CRASH COURSE AND TUTORIAL FOR EVERYONE INCLUDING DUMMIES
KEN DOUGLAS

Python Programming Cookbook for Absolute Beginners

A Complete Crash Course and Tutorial for Everyone Including Dummies

By

Ken Douglas

TABLE OF CONTENTS

CHAPTER 1

WHAT IS PYTHON?

Python is a coding language which means that can be used for other types of programming and software development besides web development.Python can be used for things like: Back end (or server-side) web and mobile app development.

PYTHON VERSION 2 VS VERSION 3

So back in 2008, the creator of Python, Guido van Rossum was a bit unhappy with some of the really important architectural decisions that were made early on in Python and he wanted to change them so he decided to overhaul Python and release Python 3.People were supposed to adopt it pretty quickly and stick with it But as you can probably tell by the fact that I have to subtitle this video it was more complicated than that and the reason for all the trouble all the controversy is that Python 3 is not backwards compatible.So usually when you have a programming language and changes are made it's made incrementally and it's made in a way that old code can still run.So old javascripts written five years ago if you tried to run it now there won't be any problems It won't be using some of the new features and It might not look as nice but it will still work.

But with Python 3 the opposite was true.Python 2 was definitely the best choice early on.But things have changed it's just been really slowly very incrementally but almost 10 years later Python 3 is absolutely the way to go.So, what makes me say that well first of all it's just easier in general to learn the most up to date current standards to learn Python 3 and then go back if you need to work in Python to for some reason.You can figure out the key differences in the quirks rather than going the other way but more importantly most of the main criticisms of things people didn't like just don't matter anymore.It used to be that all the popular packages all the tools that people would use the utilities that were written for Python were only written in Python too.So, if you upgraded to Python 3 you were in this frontier land where you couldn't rely on common packages it is no longer the case now,throughout this book we would be working with the upgraded python 3.

INSTALLING PYTHON 3

Before we begin our python journey,let us look at how we can install python on both Mac and Windows devices

MAC

STEP 1 - OPEN THE TERMINAL

We will complete most of our installation and set it up from the command line. This is a non-graphical way to interact with your computer. That is, instead of clicking buttons, type text and also get text feedback from your computer. The command line, also known as a shell, can help you change and automate many of the tasks you do on a computer every day. It is an indispensable tool for software developers.

The macOS terminal is an application that allows you to access the command line interface. As with any other application, you can find it by navigating to the Applications folder in Finder, and then to the Utilities folder. From here, as with any other application, double-click the terminal to open it. Alternatively, you can use Spotlight by holding down the "" and "Spacebar" keys to find Terminal by typing it in the box that appears.

STEP 2 - INSTALL XCODE

Xcode is an integrated development environment (IDE) that contains software development tools for macOS. You may already have Xcode installed. To verify, enter the following in your terminal window:

```
xcode-select -p
```

When you get the following output, Xcode will be installed:

```
Output/Library/Developer/CommandLineTools
```

If you get an error message, install Xcode from the App Store in your web browser and accept the default options.After installing Xcode, return to your terminal window. Next, you need to install Xcode's separate command line tools app. To do this, enter the following:

```
xcode-select --install
```

At this point, Xcode and its command line tools app are fully installed and we are ready to install the package manager homebrew.

STEP 3 - INSTALL AND SET UP HOME-BREW

Although the OS X terminal has many functions of Linux terminals and other Unix systems, it is not delivered with a good package manager. A * package manager * is a collection of software tools used to automate installation processes. This includes installing software for the first time, upgrading and configuring software, and removing software as needed. You keep the installations in a central location and can manage all software packages on the system in the formats commonly used. * Homebrew * offers OS X a free and open source system for managing software packages, which simplifies the installation of software under OS X.

Type the following in your terminal window to install homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Since Homebrew was created with Ruby, your computer's Ruby path will change. The command + curl + gets a script from the specified URL. This script explains how it works and stops the process to ask for confirmation. In this way, you receive a lot of feedback on the actions of the script on your system and have the opportunity to check the process.If you have to enter your password, note that your keystrokes are not shown in the terminal window, but are recorded, simply press the "+ Return" key as soon as you have entered your password. Otherwise, press the letter "y +" for "yes" when prompted to confirm the installation.Let's look at the flags associated with the "+ curl +" command:

1.
The flag -` + f + oder + - fail + `instructs the terminal window not to output an HTML document in the event of server errors.
2.
The + -s + or + - silent + flag + curl + mutes so that the progress bar is not displayed, and combined with the + -S + or + - show-error + flag it appears Make sure that "+ curl +" displays an error message if this fails.The"+ 3.
3.
-L" or "+ -" location and "" flags instruct "curl" to retry the request to a new location when the server reports that the requested page has been moved to another location .Once the installation process is complete, we put the homebrew directory in the environment variable "+ PATH +" above. This ensures that homebrew installations are accessed using the tools that Mac OS X automatically selects and that may conflict with the development environment we created.You should + ~ / .bash_profile + create or open the file with the command line text editor * nano * using the command + nano + :`nano ~/.bash_profile`

When the file is open in the terminal window, write the following:

```
export PATH=/usr/local/bin:$PATH
```

To save your changes, hold down the "" key and the letter "o" and press the "" key when prompted. Now you can quit Nano by holding the "" key and the letter "x".To activate these changes, enter the following in the terminal window:

```
source ~/.bash_profile
```

Once you do this, the changes you make to the "+ PATH" environment variable take effect.We can ensure that homebrew has been successfully installed by entering:

```
brew doctor
```

If no updates are required at this point, the output from the terminal is:

```
OutputYour system is ready to brew.
```

Otherwise, you may receive a warning to run another command such as "+ brew update +" to ensure that your Homebrew installation is up to date.Once Homebrew is done, you can install Python 3.

STEP 4 - INSTALL PYTHON 3

You can use homebrew to search for anything you can install using the "+ brew search +" command. However, to provide us with a shorter list, we're just looking for the available Python-related packages or modules instead:

```
brew search python
```

The terminal outputs a list of the components that can be installed:

```
Outputapp-engine-python         micropython              python3

boost-python              python                wxpython

gst-python              python-markdown           zpython

homebrew/apache/mod_python                   homebrew/versions/gst-python010

homebrew/python/python-dbusCaskroom/cask/kk7ds-python-runtime

homebrew/python/vpythonCaskroom/cask/mysql-connector-python
```

Python 3 will be one of the items on the list. Let's go ahead and install it:

```
brew install python3
```

The Terminal window gives you feedback on the Python 3 installation process. It may take a few minutes for the installation to complete.
Together with Python 3, Homebrew installs * pip *, * setuptools * and * wheel *.
As a tool for use with Python, we will use * pip * to install and manage programming packages that we may want to use in our development projects. You can install Python packages by typing:

```
pip3 install
```

Here can ++ refer to any Python package or library, e.g. B. Django for web development or NumPy for scientific computing. If you want to install NumPy, you can do this with the command + pip3 install numpy + .

1.
setuptools * makes it easy to pack Python projects, and * wheel * is an integrated package format for Python that can speed up your software production by reducing the frequency with which you have to compile.To check the version of Python 3 you have installed, you can type:

```
python3 --version
```

This will output the currently installed version of Python. By default, this is the latest stable version of Python 3 available.To update your version of Python 3, you can first update Homebrew and then Python:

```
brew update
```

```
brew upgrade python3
```

It is recommended to make sure that your version of Python is up to date.

STEP 5 - CREATE A VIRTUAL ENVIRONMENT

After installing Xcode, Homebrew and Python, we can create our programming environment.In virtual environments, you can set up an isolated area for Python projects on your computer. This ensures that each of your projects has its own dependencies that do not interfere with any of your other projects.Setting up a programming environment gives us more control over our Python projects and how different versions of packages are handled. This is especially important if you are working with third-party packages.You can set up as many Python programming environments as you want. Each environment is basically a directory or folder on your computer that contains some scripts to act as an environment.Select the directory in which you want to insert your Python programming environments, or create a new directory with + mkdir + , as in:

```
mkdir
```

```
cd
```

Once you are in the directory where you want to save the environments, you can create an environment by running the following command:

```
python3.7 -m venv
```

Essentially, this command creates a new directory (named in this case) that contains some elements:

1.
   The file "+ pyvenv.cfg +" refers to the Python installation with which you executed the command.
2.
   The subdirectory + lib + contains a copy of the Python version and contains a subdirectory + site-packages + that is initially empty, but eventually contains the third-party modules you have installed.
3.
   The subdirectory + include + compiles packages.
4.
   The "+ bin +" subdirectory contains a copy of the Python binary along with the " ACTIVATE " shell script that is used to set up the environment.

Together, these files ensure that your projects are isolated from the general context of your local computer so that system files and project files are not mixed. This is a good way to do version control and to ensure that each of your projects has access to the specific packages it needs.To use this environment, you must activate it. To do this, enter the following command that calls the activation script:

```
source /bin/activate
```

Your prompt is now preceded by the name of your environment, in this case it is called:
With this prefix we know that the environment is currently active. When we create programs here, only the settings and

packages of this particular environment are used.
After you complete these steps, your virtual environment is ready to use.

STEP 6 - CREATE A SAMPLE PROGRAM

After we set up our virtual environment, we create a traditional "Hello, World!" - program. This ensures that our environment works and gives us the opportunity to get to know Python better if it has not already been done.To do this, we open a command line text editor like nano and create a new file:

```
nano hello.py
```

As soon as the text file opens in Terminal, we enter our program:

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

Exit nano by pressing the "" and "x" buttons. When prompted to save the file, press "y +".Once you exit nano and return to your shell, run the program:

```
python hello.py
```

The hello.py program you just created should cause Terminal to produce the following output:

```
Output Hello, World!
```

To leave the environment, simply enter the command "+ deactivate +" and you will return to your original directory.

WINDOWS

As with almost every application under Windows, Python is installed using an installer that guides you through the setup process. By default, the Python installer stores .exe files under Windows in the app data directory of the respective user - so that no administrator rights are required for the installation.

FIND THE RIGHT PYTHON INSTALLER FOR WINDOWS

Python.org provides several different Windows installers. In addition to the 32-bit (x86) and the 64-bit version (x86-64) also an embeddable zip file, an .exe file and a web-based installer. These differ as follows:

1.
   The .exe installer is just an executable file that starts the Python installation process - this is the most common and simplest solution.
2.
   The web-based installer is basically identical to its .exe counterpart, with the difference that it downloads the files necessary for the installation separately. This drastically reduces the size of the installer, but a network connection is absolutely necessary.
3.
   The embeddable Zip File is a self-contained, minimalist copy of the Python run time environment. This is useful if you want to distribute a Python application manually or if you want to test something quickly. However, this installer does not contain any of the useful tools that the other versions have on board.

INSTALL PYTHON WITH A PACKAGE MANAGER ON WINDOWS

Another option under Windows is to use the NuGet operating system package management system. The Package Manager for .NET also offers Python, but as a component for .NET applications and not to be used as an installer for a standalone version of Python. Managing your Python instance should therefore be easier with a regular installation.The Windows Package Management System Chocolatey also provides Python . This is usually the better option compared to NuGet because it checks your system for an existing Python runtime environment. However, you should avoid mixing regular and Chocolatey installations on one system.
Now That we have installed python 3 let us learn some concepts!

CHAPTER 2

DATA TYPES AND VARIABLES

WHAT ARE VARIABLES?

A variable in the most general sense is simply a container (container) for storing certain values, such as strings or numbers. You can access these variables during the program, or more precisely the value of their content, or assign them a new value.
In most programming languages, such as C, a variable designates a fixed memory location in which values of a certain data type can be stored. The value of the variable can change during the program run, but the value changes must be of the same type. So you cannot have an integer stored in a variable at a certain point in time and then overwrite this value with a floating point number. The location of the variables is also constant during the entire run and cannot be changed. In languages like C, the location is already fixed by the compiler.
It looks different in Python. First of all, variables in Python do not denote a certain type and therefore you don't need a type declaration in Python either. If, for example, you need a variable i with the value 42 in the program, you can easily do this with the following statement:

```
i = 42
```

The above statement should not be seen as a mathematical equal sign, but as "the variable i is assigned the value 42", ie the content of i is after the assignment 42. You can also use this value of the variable as shown in the following example to change:

```
>>>i = 42
```

```
>>>i = i + 1
```

```
>>> print i
43
>>>
```

NUMBERS

Python knows four built-in data types for numbers:

1. Integer (integer) eg 4321 leading

0 means octal number and
leading 0x means hex number

2. long integer

They can be of any length.
They are marked with an l at the beginning or L at the end.

3. Floating point

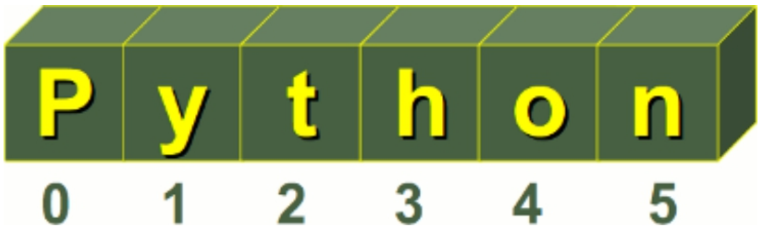
numbers 3.14159 or 17.3e + 02

4. complex numbers

e.g. 1.2 + 3j

STRINGS

A string, or string, can be seen as a sequence of individual characters.



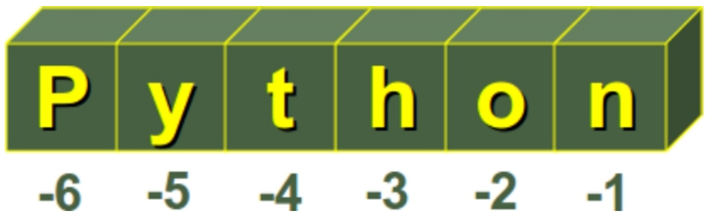
Every single character of a string can be addressed via an index. The following example shows how the above string shown in the image is defined in Python and how we can access it:

```
>>> s = "Python"
>>> print s [0]
P
>>> print s [3]
H
```

The length of a string can be determined with the len () function and you can also easily access the last or penultimate character of a string, for example:

```
>>> s = "Python"
>>>index_last_char = len (s) - 1
>>> print s [index_last_char]
n
>>> print s [index_last_char - 1]
O
>>>
```

Since it is very common in practical work that you have to access individual characters in a string from behind, it would be very annoying if you always had to do this by calling the function len (). Python therefore offers more elegant options. The indices are also numbered from the right by negative indices, ie the last character is addressed by means of the index -1, the penultimate character by means of -2 etc. We see this illustrated in the following figure:



In the code it looks like this:

```
>>> s = "Python"
>>>last_character = s [-1]
>>> print last_character
n
```

Strings can be created using

1. single quotation marks (')

' This is a string with simple quotes'

2. double quotes (")

"Mayers' Dachshund is called Waldi"

3. triple quotes (''') or (""")

''' String in triple quotes can also span multiple lines and contain 'single' and 'double' quotes. '''
can be specified

IMPORTANT STRING FUNCTIONS

A couple of string functions:

1. CONCATENATION

This function is used to connect two strings to a new string using the "+" operator:

"Hello" + "World" ->"HelloWorld"

2. REPETITION

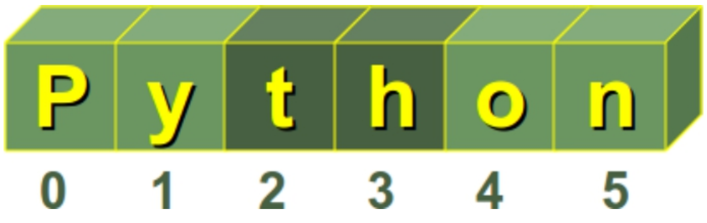
A string can be repeated concatenated. To do this, use the "*" operator.
Example:
"* - *" * 3 becomes "* - ** - ** - *"

3. INDEXING

"Python" [0] ->"P"

4. SLICING

You cut a "slice" out of a string. In the following expression, [2: 4] means that we cut out a substring from the string "Python", which begins with the character of index 2 (inclusive) and goes up to index 4 (exclusive):
"Python" [2: 4] ->"th"



5. LENGTH OF A STRING

len ("Python") -> 6

IMMUTABLE STRINGS

As in Java but not as in C or C ++, strings cannot be changed in Python. If you try to change an indexed position, an error message is generated:

```
>>> s = "Some things are immutable!"
>>> s [-1] = "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

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

ESCAPE OR ESCAPE CHARACTERS

There are strings that control the flow of text, such as a newline (line feed) or tab. They cannot be displayed as individual characters on the screen. Such characters are represented within string literals using special character strings, so-called escape sequences. An escape sequence is initiated by a backslash \ followed by the identifier of the desired special character. Overview of the escape characters:

1.
   \ Line continuation
2.
   \\ backslash
3.
   \ 'Single quotation mark
4.
   \ "Double quotes
5.
   \ a bell
6.
   \ b regression
7.
   Mask out
8.
   \ 0 zero
9.
   \ n line feed (LF)
10.
    \ v Vertical tab

11.
\ t Horizontal tab

12.
\ r Carriage return (CR)

13.
\ f form feed

14.
\ 0XX Octal value

15.
\ xXX hexadecimal value

The evaluation of escape characters can be prevented by placing an r or R in front of a string.

Example:

r "\ n causes a line feed"

TYPE CHANGE FOR VARIABLES

In Python, as already mentioned, a variable can be used immediately without declaring the data type. Nevertheless, Python assigns a data type, ie depending on the data type, the variable is created differently, i.e. as an integer, float, string, and so on. One should actually say that an object is created with a certain data type or class. The variable then references this object, ie the variable itself is actually of no type. In other words,

the data type in Python is not bound to the variable, but to the value, which implies that the type can change at run-time, as we can see in the following example:

```
i = 42 # data type is integer (implicit)
```

```
i = 42 + 0.11 # type changes to float
```

```
i = "fourty" # and now a string
```

CHANGING LOCATIONS

In principle, the storage location for the variable i will change in the previous case, although this of course depends on the implementation. With the instruction "i = 42" the interpreter can save the value as an integer, but with the instruction "i = 42 + 0.11" it has to create a new location for a float number. For i = "forty" it must be converted into a string.

Attention: As a user, you don't actually need to know this, because everything happens automatically!

Now let's consider the following Python code:

```
>>> x = 3
>>> y = x
>>> y = 2
```

Intuitively, one would assume that Python first chooses a storage location for x and stores the object (number) 3 there. The variable y is then assigned the value of x. In C and many other programming languages there would also be a separate storage location for y, in which the number 3 would now be written. Python does it differently: x is a variable with object 3 and y is a variable with the "same" (not "same") object. x and y "point" to the same object. In the last line, y is now assigned the value 2, now a new object must be created and y "points" to a new storage location. (Note: This "show" just used should not be confused by C programmers with the pointers used under C.)

The question now is how to check the above. The identity function id () is useful for this. The identity of an instance is used to distinguish it from all other instances. The identity is an integer and it is unique within a program. The identity function id () provides the identity. So you can check whether it is a specific instance and not just one with the same value and type. We enter the above example again, but let us output the identity in each case:

```
>>> x = 3
>>> print id (x)
157379912
>>> y = x
>>> print id (y)
157379912
>>> y = 2
>>> print id (y)
157379924
>>> print id (x)
157379912
>>>
```

We find that the identity only changes after we assign a new value to y. The identity of x remains the same, ie the location of x is not changed.

SPECIAL FEATURES OF STRINGS

```
>>> a = "Linux"
>>> b = "Linux"
>>> a is b
True
```

But how does it look if the string used is longer? In the following we use the longest place name in the world as a string. A parish with just over 3000 inhabitants in the south of the island of Anglesey in the county of the same name in northwest Wales:

```
>>> a = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
>>>                                  b                                  =
"Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
>>> a is b
True
```

But caution is advised, because what works for a community in Wales, for example, fails for Baden-Württemberg:

```
>>> a = "Baden-Württemberg"
>>> b = "Baden-Württemberg"
>>> a is b
False
>>> a == b
True
```

So it cannot be because of the geographical location, as you can see in the following example. It looks like there are no special characters or blanks in the string.

```
>>> a = "Bathing!"
>>> b = "Bathing!"
>>> a is b
False
>>> a = "Baden1"
>>> b = "Baden1"
>>> a is b
True
```

CHAPTER 3

OPERATORS AND DEEP/FLAT COPYING

An expression in Python and in other programming languages is a combination of variables, constants, operators and return values of functions. The evaluation of an expression gives a value that is usually assigned to a variable. In Python, expressions are written using common mathematical notations and symbols for operators.

OPERATORS

Most numeric value operators in Python are similar to other programming languages. We give an overview here without fully explaining it. If necessary, these operators are discussed in other chapters.

operator	Designation	example
+, -	Addition, subtraction	10 - 3
*, /,%	Multiplication, division, remainder	27 % 7 result: 6
+ x, -x	Sign	-3
~ x	Bit by bit emergency	~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]
<, <=,>,>=,!=, ==	The usual comparison operators	2 <= 3

|, &, ^	Bitwise or, bitwise and, bitwise XO R	6 ^ 3
<<, >>	S h i f t o p e r a t o r s	6 < < 3

DEEP AND FLAT COPYING

As we saw in the last chapter "Data Types and Variables", Python behaves unusual when copying simple data types like integers and strings compared to other programming languages.

In the following code example, y initially only points to the same memory location as x. Only when the value of y is changed does y get its own storage space, as we saw in the previous chapter.

```
>>> x = 3
>>> y = x
```

But even if the above behavior is unusual compared to other programming languages such as C, C ++, Perl and others, the results of the assignments still meet our expectations. However, it becomes critical if we want to copy mutable objects such as lists and dictionaries. Python only creates real copies if it absolutely has to, ie that the user, i.e. the programmer, explicitly requests it. In this chapter we want to show some problems that can arise when copying mutable objects, for example when copying lists and dictionaries.

COPY A LIST

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 = ["rouge", "vert"]
>>> print colors1
['red', 'green']
```

In the code example above, we first create a simple list colours1, which we then copy into colours2. Then we assign a new list to colours2.

It is not surprising that the values of colours1 remain unchanged. As with the example with the integer variables in the last chapter "Data types and variables", a new memory area is created for colours2 when a completely new list is assigned to this variable.

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 [1] = "blue"
>>> colours1
['red', 'blue']
```

But what does it look like if only individual elements are changed?

To test this, we assign a new value to the second element of colours2. Many will now be amazed that colours1 has also been changed, although it was believed that they had made a copy of colours1. The explanation is that the associated object has not been changed by colours2.

COPY WITH SECTION OPERATOR

With the section operator (slicing) you can completely copy flat list structures without side effects, as can be seen in the following example:

```
>>> list1 = ['a', 'b', 'c', 'd']
>>> list2 = list1 [:]
>>> list2 [1] = 'x'
>>> print list2
['a', 'x', 'c', 'd']
>>> print list1
['a', 'b', 'c', 'd']
```

```
>>>
```

However, as soon as sub-lists appear in the list to be copied, only pointers to these sub-lists are copied.

```
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>> lst2 = lst1 [:]
```

If, for example, the 0th element of one of the two lists is assigned a new value, this does not lead to a side effect. Problems only arise if you change one of the two elements of the sub-list directly.

To demonstrate this, we are now changing two entries in lst2

```
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>> lst2 = lst1 [:]
>>> lst2 [0] = 'c'
>>> lst2 [2] [1] = 'd'
>>> print (lst1)
['a', 'b', ['ab', 'd']]
>>>
```

You can see that you have not only changed the entries in lst2, but also the entry of lst1 [2] [1].

This is because in both lists, ie lst1 and lst2, the third element is only a link to a physically identical sublist. This sublist was not copied with [:].

COPY USING THE DEEP COPY METHOD FROM THE COPY MODULE

The module "copy" solves the problem just described. This module provides the "deepcopy" method, which allows the complete copying of a non-flat list structure.

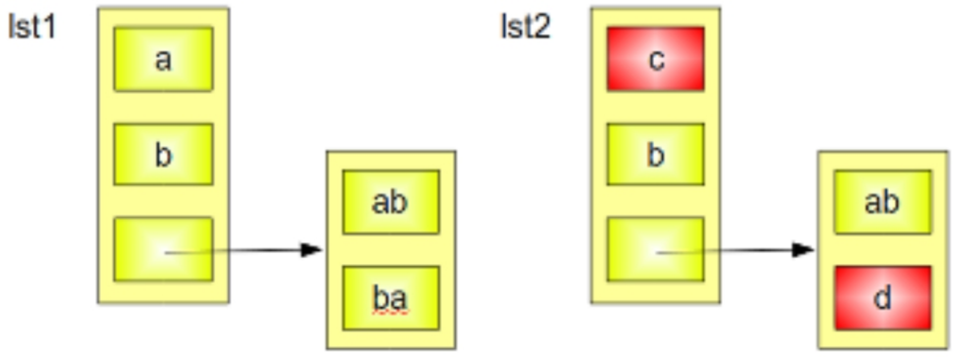
The following script copies our example above using this method:

```
from copy import deepcopy
lst1 = ['a', 'b', ['ab', 'ba']]
```

```
lst2 = deepcopy (lst1)
lst2 [2] [1] = "d"
is2 [0] = "c";
print lst2
print lst1
```

If we save the script under deep_copy.py and call it up with "python deep_copy.py", we get the following output:

```
$ python deep_copy.py
['c', 'b', ['ab', 'd']]
['a', 'b', ['ab', 'ba']]
```



CHAPTER 4

CONDITIONAL STATEMENTS

At certain times, certain decisions are inevitable, as you can see in the photo. So, you can hardly write a meaningful program that works without any branching. So far, we have been able to write programs in which one instruction follows the other and these are also executed in this order. But only a few problems can be controlled by a linear program flow. For example, you only want to execute a certain part of the program if certain conditions apply or another part should possibly be executed several times. Each programming language offers control structures that can be divided into two categories: branches and loops.

In a programming language, a conditional statement, as it is called a branch, is a piece of code that is executed under certain conditions. If the condition is not met, this code is not executed. In other words: a branch determines which of two (or more) program parts (alternatives) is executed depending on one (or more) conditions. Conditional instructions and branches are assigned to the control structures in programming languages (just like the loops), because with their help a program can react to various states that result from inputs and calculations.

The final example in this chapter is about taxes. We are writing a python script that can be used to calculate the taxes for 2010 from the taxable income.

THE IF STATEMENT

The general form of the if statement in Python looks like this:

```
if condition1:
    instructions 1
```

```
elif condition2:
   instructions2
else:
   instructions 3
```

If the condition "condition1" is true, the instructions "instructions1" are executed. If not, if condition2 is true, instructions2 are executed. If neither the first condition (condition1) nor the second condition (condition2) is true, the statements after the else (statements3) are executed.

EXAMPLE DOG YEARS

Children and dog lovers often ask how old their dog would be if it were not a dog but a human. It is common practice to convert dog years into human years by multiplying the dog's age by 7. Depending on the dog size and breed, the conversion may look a little more complicated, for example:

A one-year-old dog roughly corresponds to a 14-year-old human

2 years of a dog corresponds to 22 years of a human.

From then on, a dog year corresponds to 5 human years.

The following small example program requires input for the age of the dog and calculates the age in human years according to the above rule:

```
age = input ("age of the dog:")
print
if age <0:
        print "That is hardly true!"
elif age == 1:
        print "corresponds to approx. 14 years"
elif age == 2:
        print "corresponds to approx. 22 years"
elif age> 2:
        human = 22 + (age -2) * 5
        print "human years", human
###
raw_input ('press return>')
```

TRUE OR FALSE

Unfortunately, it is not so easy to distinguish between true and false in all things in life as in Python: to be

considered "false"

numeric zero values (0, 0L, 0.0, 0.0 + 0.0j),

the boolean value false,

empty strings,

empty lists, empty tuples,

empty dictionaries.

and the special value None.

Python considers all other values "true".

ABBREVIATED IF

C programmers know the following abbreviation for the if construct:

```
max = (a> b)? a: b;
```

as an abbreviation for:

```
if (a> b)
  max = a;
```

```
else
  max = b;
```

In Python, C programmers have to get used to this spelling. In Python the previous example is formulated as follows:

```
max = a if (a> b) else b;
```

CONTROL CALCULATOR IN PYTHON

The taxable income (zvE) is initially assigned to a tariff zone. Then the tax amount (StB) can be calculated according to the corresponding formula for individual assessment. 1

1. First zone (basic allowance): up to € 8004, there is no tax
2. Second zone: zvE from € 8,005 to € 13,469

StB = (912.17 * y + 1400) * y

For y:

y = (zvE - 8004) / 10000

3. Third zone: zvE from € 13470 to € 52881

StB = (228.74 * z + 2397) * z

For z:

z = (zvE - 13469) / 10000

4. Fourth zone: zvE from € 52882 to € 250730

StB = 0.42 * zvE - 8172

5.
Fifth zone: zvE from € 250,731

StB = 0.44 * zvE - 15694

A Python function for calculating the tax from the income tariff now looks like this: (of course without guarantee !!!)

```
def taxes (income):
"""Calculation of taxes payable for taxable income x"""
    if income <= 8004:
        tax = 0
elif income <= 13469:
        y = (income -8004.0) /10000.0
        tax = (912.17 * y + 1400) * y
elif income <= 52881:
        z = (income -13469.0) /10000.0
        tax = (228.74 * z +2397.0) * z +1038.0
elif income <= 250730:
        tax = income * 0.42 - 8172.0
    else:
        tax = income * 0.44 - 15694
    return tax
```

In the case of spouse splitting, the taxes can be calculated with the following call:

```
taxes = 2 * taxes (income / 2)
```

So that you can still calculate your taxes in 2013 with Python, here you will find the tax calculation for 2013: (No guarantee)

Corresponding German legislative text:

Income Tax Act, Section 52 Application

Regulations (41) Section 32a (1) shall apply from the 2010 assessment period in the following version:

"(1) 1 The collective income tax is based on the taxable income. 2 Subject to the Sections 32b, 32d, 34, 34a, 34b and 34c each in euros for taxable income

1. up to 8 004 euros (basic allowance):

0;

6.
from 8 005 euros to 13 469 euros:

(912.17 * y + 1 400) * y;

3. from 13 470 euros to 52 881 euros:

(228.74 * z + 2 397) * z + 1 038;

4. from 52 882 euros to 250 730 euros:

0.42 * x - 8 172;

6.
from 250 731 euros to:

0.45 * x - 15,694.

Nothing changed in principle in the formulas for calculating income tax in 2013. The changes are highlighted in red below:

1. up to 8 354 euros (basic allowance):

0;

7.
from 8 355 euros to 13 469 euros:

(912.17 * y + 1 400) * y;

3. from 13 470 euros to 52 881 euros:

(228.74 * z + 2 397) * z + 1 038;

4. from 52 882 euros to 250 730 euros:

0.42 * x - 8 239 ;

5. from 250 731 euros to:

0.45 * x - 15 761 .

3 "y" is a ten-thousandth of the part of the taxable income that is rounded up to a full euro amount and exceeds 8,004 euros. 4 "z" is a ten-thousandth of the portion of the taxable income that is rounded up to a full amount of euro, exceeding EUR 13 469. 5 "x" is taxable income rounded off to the nearest euro. 6 The resulting tax amount must be rounded down to the nearest full euro amount. "

CHAPTER 5

INPUTS AND LOOPS

There are hardly any programs that can do without any input. Entries can be made in many ways, for example from a database, from another computer in the local network or via the Internet. The simplest and probably the most common input is made using the keyboard. For this form of input, Python offers the function input (text).

If the input function is called up during a program run, the program sequence is stopped until the user makes an entry via the keyboard and closes it with the return key. So that the user knows what to enter, the string of the parameter "text" is output if such a string exists. The input () parameter is optional.

The input string of the user is interpreted by input (), ie input () returns, for example, an integer if the user has entered an integer and a list if the user has entered a list.

We show this in the following interactive Python shell session:

```
>>> x = input ("your name?")
Your name? "John"
>>> print (x)
John
>>> x = input ("your salary?")
Your salary? 2877.03
>>> x
2877.03
>>> type (x)
<type 'float'>
```

```
>>> x = input ("your favorite languages?")
Your favorite languages? ["Java", "Perl", "C ++"]
>>> print (x)
['Java', 'Perl', 'C ++']
>>> type (x)
<type 'list'>
>>>
```

INPUT WITH RAW_INPUT ()

Unlike input, raw_input does not interpret the input. This means that raw_input always returns a string, ie the input string of the user is passed on unchanged. If you want a certain data type, you can convert the input using the corresponding casting function or you can use the eval function. We demonstrate this again with some examples in the interactive Python shell:

```
>>> x = raw_input ("your name?")
Your name? John
>>> print (x)
John
>>> x = raw_input ("your salary?")
Your salary? 2877.03
>>> print (x)
2877.03
>>> type (x)
<type 'str'>
>>> x = float (raw_input ("your salary?"))
Your salary? 2877.03
```

```
>>> type (x)
<type 'float'>
>>> x = eval (raw_input ("your favorite languages?"))
Your favorite languages? ["Java", "Perl", "C ++"]
>>> print (x)
['Java', 'Perl', 'C ++']
>>> type (x)
<type 'list'>
>>>
```

LOOPS

GENERAL STRUCTURE OF A LOOP

Loops are required to repeatedly execute a block of code, which is also called a loop body. There are two types of loops in Python: the while loop and the for loop.Most loops contain a counter or, more generally, variables that change their values as the calculations inside the loop body. Outside, ie before the start of the loop, these variables are initialized. Before each loop it is checked whether an expression in which these variables or variables occur is true. This expression determines the end criterion of the loop. As long as the calculation of this expression returns "True", the body of the loop is executed. After all instructions of the loop body have been carried out, the program control automatically jumps back to the beginning of the loop, i.e. to check the end criterion, and checks again whether this has been fulfilled again.If so, it continues as described above, otherwise the loop body is no longer executed and the rest of the script is continued. The diagram opposite shows this schematically.

SIMPLE EXAMPLE OF A LOOP

We would now like to illustrate this with a small Python script, which forms the sum of the numbers from 1 to 100. In the next chapter on for loops we will learn about a far more elegant way to do this.

```
n = 100
s = 0
i = 1
while i<= n:
   s = s + i
i = i + 1
print "The total is:", p
```

READ STANDARD INPUT

Before we continue with the while loop, we need to clarify a few basic things about standard input and standard output. The keyboard is normally the standard input. Most shell programs write their output to standard output, ie the terminal window or the text console. Error messages are output in the standard error output, which usually also corresponds to the current terminal window or the text console.

The Python interpreter also provides three standard file objects:

1. Standard input
2. Standard edition
3. Standard error output

They are in the sys module as

1. sys.stdin
2. sys.stdout
3.

    sys.stderror

to disposal.

The following example script now shows how to read in character by character from the standard input (keyboard) using a while loop. The required sys module is read in with the import command.

```
import sys
text = ""
while 1:
  c = sys.stdin.read (1)
  text = text + c
  if c == '\ n':
     break
print "Entry:% s"% text
```

Of course, you can read any input line from standard input more elegantly with the function raw_input (prompt).

```
>>> name = raw_input ("What's your name? \ n")
What's your name?
Tux
>>> print name
Tux
>>>
```

THE ELSE PART

Like the conditional if statement, the while loop in Python, unlike other programming languages, has an optional else branch, which many programmers need to get used to.

The instructions in the else part are executed as soon as the condition is no longer met. Certainly some are wondering what is the difference to a normal while loop. If you hadn't put the statements in the else part but simply put them behind the while loop, they would have been executed exactly the same way. It only makes sense with a break command, which we will get to know later.

In general, a while loop with else part looks like this in Python:

```
while condition:
        Instruction 1
        ...
        Instructions
else:
        Instruction 1
        ...
        Instructions
```

PREMATURE TERMINATION OF A WHILE LOOP

A loop is normally only ended when the condition in the loop head is no longer fulfilled. With break you can leave a loop prematurely and end a run with continue.

In the following example, a simple number guessing game, mam can see that in combination with a break the else branch can make sense. Only when the while loop ends regularly, ie the player has guessed the number, is there a congratulation. If the player gives up, ie break, the else branch of the while loop is not executed.

```
import random
n = 20
to_be_guessed = int (n * random.random ()) + 1
guess = 0
while guess! = to_be_guessed:
   guess = input ("New number:")
   if guess> 0:
    if guess>to_be_guessed:
        print "Number too large"
elif guess <to_be_guessed:
        print "Number too small"
  else:
    print "Too bad you give up!"
    break
else:
  print "Congratulations! That's it!"
```

FOR LOOP SYNTAX

```
for variable in sequence:
      Instruction 1
      Instruction2
      ...
      Instructions
else:
      Else Instruction 1
      Else statement2
      ...
      Else statement
```

The for statement has a different character from the for loops, which are known from most other programming languages. In Python, the for loop is used to iterate over a sequence of objects, whereas in other languages it is usually just "a slightly different while loop".

Example of a for loop in Python:

```
>>> languages   = ["C", "C ++", "Perl", "Python"]
>>> for x in languages:
```

```
... print x
...
C.
C ++
Perl
python
>>>
```

THE RANGE () FUNCTION

With the help of the range () function, the for loop can be used ideally for iterations. range () returns lists that correspond to arithmetic enumerations.
Example:

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

The above example shows that Range with an argument returns the list of numbers from 0 to this argument.
range () can also be called with two arguments:

```
range (begin, end)
```

A list of all integers from begin (inclusive) to end (exclusive) is then provided.
As a third argument, range () can also be given the step size.
Example:

```
>>> range (4.10)
[4, 5, 6, 7, 8, 9]
>>> range (4,50,5)
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```

The range () function is particularly useful in conjunction with the for loop. In the following example we form the sum of the numbers from 1 to 100:

```
n = 100
s = 0
for i in range (1, n + 1):
    s = s + i
print p
```

In the small program above, there is still a terrible problem with efficiency. What happens before the for loop is executed? Python first evaluates the range (1, n + 1) call. This means that a list with 100 numbers is generated, ie [1, 2, 3, 4, ... 100]. All the numbers in this list are required within the loop, but the entire list is never required. In the previous chapter we solved this problem with a while loop and we didn't need a list there. Python offers a solution to this problem by providing the xrange function. xrange creates an iterable object, which means that no list is created, but can be iterated over, for example, in a for loop via the values without the list being generated:

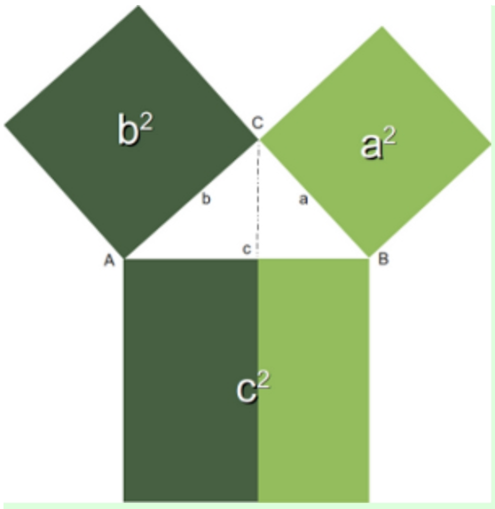
```
>>> for i in xrange (1, 7):
... print (i)
...
1
2nd
3rd
4th
5
6
>>>
```

The above loop behaves similarly to the following while loop in terms of efficiency:

```
>>>i = 1
```

```
>>> while i<7:
... print (i)
... i + = 1
...
1
2nd
3rd
4th
5
6
>>>
```

Of course, there is no difference in the output behavior. The difference between range and xrange can be seen, however, if you make the calls directly in the interactive Python shell:



```
>>> range (1.7)
[1, 2, 3, 4, 5, 6]
>>>xrange (1.7)
xrange (1, 7)
>>>
```

EXAMPLE: CALCULATING THE PYTHAGOREAN NUMBERS

Most believe that the Pythagorean theorem was discovered by Pythagoras. Why else would the sentence have got its name? But there is a debate as to whether this sentence could not have been discovered independently of Pyhtagoras and, above all, earlier. For the Pythagoreans - a mystical movement based on mathematics, religion and philosophy - the whole numbers that fulfilled the Pythagorean theorem were special numbers that were sacred to them.

Nowadays the Pythagorean numbers are no longer mystical. Although it may still appear that way to some students or other people who are at war with mathematics.

In mathematics, the following is very unromantic:

Three natural numbers that satisfy the equation a 2 + b 2 = c 2 are called Pythagorean numbers.

The following program calculates all Pythagorean numbers up to a maximum number to be entered:

```
#! / usr / bin / env python
from math import sqrt
n = raw_input ("Maximum number?")
n = int (n) +1
for a in xrange (1, n):
   for b in xrange (a, n):
c_square = a ** 2 + b ** 2
      c = int (sqrt (c_square))
      if ((c_square - c ** 2) == 0):
         print a, b, c
```

ITERATION OVER LIST WITH RANGE ()

If you want to access the indexes of a list, it doesn't seem to be a good idea to use a for loop to iterate over the list. You can then reach all elements, but the index of an element is not available. But there is a way to access both the index and the element. The solution is to use range () in combination with the len () function, which gives you the number of list elements:

```
fibonacci = [0,1,1,2,3,5,8,13,21]
for i in xrange (len (fibonacci)):
    print i, fibonacci [i]
print
```

LIST ITERATION WITH SIDE EFFECTS

If you iterate over a list, you should avoid changing the list in the body. The following example shows what can happen:

```
colors = ["red"]
for i in colors:
    if i == "red":
        colors + = ["black"]
    if i == "black":
        colors + = ["white"]
print colors
```

What does the "print colors" statement output?

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

It is best to use a copy of the list, as in the next example:

```
colors = ["red"]
for i in colors [:]:
     if i == "red":
        colors + = ["black"]
```

```
    if i == "black":
        colors + = ["white"]
print colors
```

The output now looks like this:

```
['red', 'black']
```

Even now we have changed the list, but "consciously" within the loop body. But the elements that are interposed through the for loop remain unchanged through the iterations.

CHAPTER 6

PRINT AND OUTPUT

There are actually no computer programs and of course no Python programs that do not communicate with the outside world in any way. Above all, a program must always output results. One form of outputting results goes directly into standard output and this is done in Python using the print statement.

```
>>> print "Hello user"
Hello user
>>> answer = 42
>>> print "The answer is:" + str (answer)
The answer is 42
>>>
```

As of Python3, print is no longer an instruction, but a function. Therefore, the arguments must be written in brackets. In Python2 you can also write print with brackets:

```
>>> print ("hello")
Hello
>>> print ("Hello", "Python")
('Hello', 'Python')
>>> print "Hello", "Python"
Hello Python
>>>
```

We can see that there is a change in output behavior. But what is more serious. The output behavior with parentheses in Python 2.x is also different from that of the Python function of version 3.x, as we can see below:

```
$ python3
Python 3.2.3 (default, Apr 10 2013, 05:03:36)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("hello")
Hello
>>> print ("Hello", "Python")
Hello Python
>>>
```

If you want to have the same output behavior as from 3.x in version 2.x, an import from the future is recommended:

IMPORT FROM THE FUTURE: PRINT_FUNCTION

The following import line can be found in some Python2 programs:

```
from __future__ import print_function
```

At first glance, this is misleading. It looks like you imported a function called "print_function". However, only one flag, an instance "print_function", was set. This introduces the Python3 print function in Python2. This means that the interpreter then only accepts the Python3 syntax of the print function. The program behaves, at least as far as the print function is concerned, compatible with Python3.
If you want to create programs in Python 2.6 or Python 2.7, we recommend that you always provide them with the above import. This ensures that your output is compatible with Python 3. At this point we therefore refer to the introduction to the print function under Python3 from our tutorial.

Notes:
In Python3, print is no longer a statement, but a function.

DIFFERENT WAYS TO FORMAT THE OUTPUT

When you learn to program, the focus is initially on the programming logic. You want to quickly learn how variables, data structures, branches, loops, etc. work. The "beautiful" output of the results is secondary for now. The only important thing is whether the right thing comes out, whether beautifully formatted or not. But at some point, the learner's desire for a nice output of the results also comes into focus.

In this chapter of our Python tutorial, we will deal intensively with the various methods with which you can format the output or generate formatted strings. We introduce the different types, but we recommend using the format method of the string class, which is just before the end of the chapter. The format method is by far the most flexible.

CHAPTER 7

SEQUENTIAL DATA TYPES

A sequential data type is a data type that contains a sequence of similar or different elements. The elements of a sequential data type have a defined order and can therefore be accessed via indices.

Python provides the following data types:

STRINGS

str

This type stores byte strings, also pure ASCII strings

unicode

This data type stores strings in Unicode.

LISTS

A sequence of arbitrary instances can be saved in a list. A list can be changed at any time while the program is running.

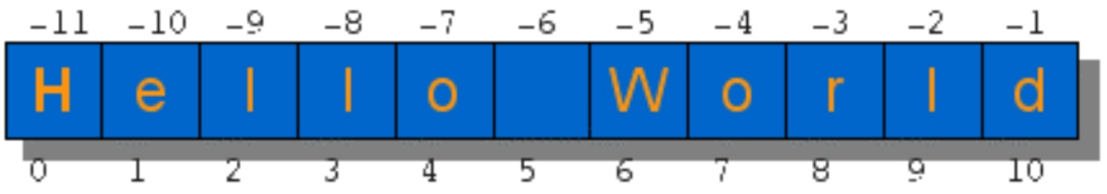
TUPLE

in a tuple is like a list stored in a sequence believer instances, but these can then no longer be changed during the further course of the program.

The operations discussed below apply to all sequential data types

INDEXING

Let's look at the string "Hello World":



You can see that the characters in a string are numbered from left to right, starting with 0. From the back (right) you start counting with -1. Each character of a string can be addressed so clearly, they can be indexed with square brackets, as can be seen in the following example:

```
>>> txt = "Hello World"
>>> txt [0]
'H'
>>> txt [4]
'O'
```

Instead of from the front, you can also determine the indices from behind:

```
>>> txt [-1]
'd'
>>> txt [-5]
'W'
```

This works the same for lists and tuples, but first we have to take a look at what lists and tuples look like in Python:
lists can contain any Python objects. In the following example, "colors" contains 3 strings and list a contains 4 objects, namely the strings "Swen" and "Basel", the integer 45 and the floating point number 3.54. If you now access a list element with an index analogously to the procedure for strings, you get the respective element. For example, colors [0] returns the string 'red' as the result.

```
>>> colors = ['red', 'green', 'blue']
>>> colors [0]
'red'
>>> colors
 ['red', 'green', 'blue']
>>>
>>>
>>> a = ["Swen", 45, 3.54, "Basel"]
>>> a [3]
'Basel'
>>>
```

SLICING

You can also cut out parts of a sequential data type. In the case of a string, you get a sub string or a list again for lists. In English this cutting is called "slicing". As with indexing, the slicing operator uses square brackets, but now at least two values are expected instead of a value: start value and end value
This is best understood using an example:

```
>>> txt = "Hello World"
>>> txt [1: 5]
'ello'
>>> txt [0: 5]
'Hello'
>>> txt = "Hello World"
>>> txt [0: 5]
'Hello'
```

If you omit the initial value (eg [: 5]), the cut begins at the beginning of the string (or the list). Similarly, you can also omit the end value to apply everything to the end (eg [6:])
If you omit the start and end values, you get the whole string (or the entire list or tuple):

```
'Hello'
>>> txt [0: -6]
'Hello'
>>> txt [: 5]
'Hello'
>>> txt [6:]
'World'
>>> txt [:]
'Hello World'
```

The following example shows how this affects lists:

```
>>> colors = ['red', 'green', 'blue']
>>> colors [1: 3]
['green', 'blue']
>>> colors [2:]
['blue']
>>> colors [: 2]
['red', 'green']
>>>
>>> colors [-1]
'blue'
```

The above slicing operator also works with three arguments. The third argument then specifies how many arguments should be taken each time, ie s [begin, end, step].

The following elements of s are then output: s [begin], s [begin + 1 * step], ... s [begin + i * step] as long as (begin + i * step) <end.
txt [:: 3] prints every third letter of a string.
Example:

```
>>> txt = "Python is really great"
>>> txt [2: 15: 3]
'tnsgz'
>>> txt [:: 3]
'Ph ta l'
```

SUB-LISTS

LISTS CAN ALSO CONTAIN OTHER LISTS AS ELEMENTS:

```
>>> pers = [["Marc", "Mayer"], ["Hauptstr. 17", "12345", "Musterstadt"], "07876/7876"]
>>> name = pers [0]
>>> name [1]
'Mayer'
>>> address = pers [1]
>>> address [1]
'12345'
>>> pers [2]
'07876/7876'
>>> street = pers [1] [0]
>>> street
'Hauptstr. 17 '
>>>
```

LENGTH

The length of a sequential data type corresponds to the number of its elements and is determined with the len () function.

```
>>> txt = "Hello World"
>>>len (txt)
11
>>>
```

Works exactly the same with lists:

```
>>> a = ["Swen", 45, 3.54, "Basel"]
>>>len (a)
4th
```

SEQUENCE CHAINING

A useful and frequently required operation on sequences is concatenation. The + sign serves as the operator sign for the chaining. The following example concatenates two strings into one:

```
>>>firstname = "Homer"
>>> surname = "Simpson"
>>> name = firstname + "" + surname
>>> print name
Homer Simpson
>>>
```

This is just as easy for lists, as the following self-explanatory example shows:

```
>>> colours1 = ["red", "green", "blue"]
>>> colours2 = ["black", "white"]
>>> colors = colors1 + colors2
>>> print colors
['red', 'green', 'blue', 'black', 'white']
```

A very common method of chaining is the "+ =", which is used in many other programming languages, especially for numerical assignments. The + = operator is used as an abbreviation, it says

```
s + = t
```

for the instruction:

```
s = s + t
```

CHECK WHETHER ELEMENT IS IN SEQUENCE

With sequences you can also check whether (or not) an element is present in a sequence. There are operators "in" and "not in" for this. The way of working can be seen in the following minutes of an interactive session:

```
>>>abc = ["a", "b", "c", "d", "e"]
>>>"a" in abc
True
>>>"a" not in abc
False
>>>"e" not in abc
False
>>>"f" not in abc
True
>>> str = "Python is great!"
>>>"y" in st
True
>>>"x" in st
False
>>>
```

REPETITIONS

A product is also defined for sequences in Python. The product of a sequence s with an integer value n (s * n or n * s) is defined as n-times concatenation of s with itself.

```
>>> 3 * "xyz-"
'xyz-xyz-xyz-'
>>>"xyz-" * 3
'xyz-xyz-xyz-'
>>> 3 * ["a", "b", "c"]
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

s * = n is (as usual) the short form for s = s * n.

THE PITFALLS OF REPETITIONS

In the previous examples, we only used the repeat operator on strings and flat lists. But we can also apply it to nested lists:

```
>>> x = ["a", "b", "c"]
>>> y = [x] * 4
>>> y
[['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b ',' c ']]
>>> y [0] [0] = "p"
>>> y
[['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b ',' c ']]
>>>
```

The result is very amazing for beginners of Python programming. We have assigned a new value to the first element of the first sub-list (i.e. y [0] [0]) and at the same time we have automatically changed the respective first element of all other sub-lists, i.e. y [1] [0], y [2] [0], y [3] [0]

The reason for this apparently strange behavior is that the repeat operator "* 4" creates four references to the list x.

CHAPTER 8

LISTS

In this chapter, we want to cover other aspects of lists. The main thing is to add, insert and delete elements.

A list can be viewed as a stack. In computer science, a stack (or stack) - sometimes also called a cellar - describes a data structure with at least two operations: one that can be used to put data on the stack and one to remove the top element of the stack . Think of it as a stack of books to read. You want to read the books in the stack from top to bottom. If you buy a new book in between, it will appear on top of it and is the next book to be read. The following operations are usually provided in the programming languages:

PUSH

This method is used to put a new object on the stack. Depending on the point of view, the object "top" or "right" is added.

POP

This method returns the top object of a stack.

PEEK

This method is used to check what is on the top of the stack. However, the object is not removed as with pop. There is no such method in Python. With lists or tuples, however, you can simulate them with index access. If "list" is a list, then list [-1] behaves like a method list.peek (), which is not available in the list class.

POP AND APPEND

1.
s.append (x) Appends

x to the end of the list s. This corresponds to the "push" method as found in other programming languages such as Perl.

2.
s.pop (i)

Returns the i-th element of s, removing it from the list. Usually, in other languages, pop only returns the "top" or the rightmost element.

3.
s.pop ()

If i is not specified, the last element is taken, which corresponds to the usual pop of other programming languages.

```
>>> l = [42,98,77]
>>>l.append (103)
>>> l
[42, 98, 77, 103]
>>> x = l.pop ()
>>> x
103
>>>l.pop ()
77
>>>
```

You quickly find yourself in the situation where you want to add more than one element to a list. For example, you want to append the elements to a list. If you try this with append, you experience an unpleasant surprise:

```
>>> l = [42,98,77]
>>> l2 = [8.69]
>>>l.append (l2)
>>> l
[42, 98, 77, [8, 69]]
>>>
```

We actually "expected" this result:

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

EXTEND

For these cases there is the extend method for lists. It is used to add several elements to a list:

```
>>> l = [42,98,77]
>>> l2 = [8.69]
>>>l.extend (l2)
>>> l
[42, 98, 77, 8, 69]
>>>
```

The extend argument must be an iterable object. For example, extend can also be used on tuples and strings:

```
>>> l = [42,98,77]
>>>l.extend ("hello")
>>> l
[42, 98, 77, 'h', 'e',   'l', 'l', 'o']
>>>
>>> l = [42,98,77]
>>>l.extend ((3,4,5))
```

```
>>> l
[42, 98, 77, 3, 4, 5]
>>>
```

THE '+' OPERATOR AS AN ALTERNATIVE TO APPEND

In addition to append, there are other options for adding elements to a list. For example, you can add one or more elements to a list with the "+" operator:

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

As far as run time behavior is concerned, extreme caution is required with this approach, as we will see below. Another option is to use the augmented assignment, compound assignment:

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

Logically speaking, both approaches are equivalent, ie they deliver the same results. In the following we want to look at the run time behavior of these two and the append method in comparison. We measure the run time using the time module, which we will not go into further here. To understand the program, it is sufficient to know that time.time () returns a float number that represents the time in seconds since "The Epoch" [1] . With time.time () - start_time we calculate the time in seconds that was necessary to calculate 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)
start_time = time.time ()
l = []
for i in range (n):
    l + = [i * 2]
print (time.time () - start_time)
start_time = time.time ()
l = []
for i in range (n):
l.append (i * 2)
print (time.time () - start_time)
```

This program gives "terrifying" results:

```
26.3175041676
0.0305399894714
0.0207479000092
```

T he "+" operator is about 1268 times slower than the append method in this run. The explanation is simple: With the append method, an additional element is simply added to the list in each loop pass. In the first case, the complete list is copied in each loop pass, ie with every assignment l = l + [i * 2], and then the new element is appended to the new list. Then the storage space for the old list - which is no longer required - must be released. We can also see that using the extended assignment in the second case is almost as fast as the way with append compared to the first method. However, the extended method is still a bit slower, since with append, the transferred object is only referenced.

FIND THE POSITION OF AN ITEM

The index method can be used to determine the position of an element within a list.

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

The index for the x is determined. If the optional parameter i is given, the search only starts from this position and ends at position j if j is given. An error message is issued if x does not occur in s.

```
>>> colors = ["red", "green", "blue", "green", "yellow"]
>>>colours.index ("green")
1
>>>colours.index ("green", 2)
3rd
>>>colours.index ("green", 3.4)
3rd
>>>colours.index ("black")
Traceback (most recent call last):
  File "", line 1, in
ValueError: 'black' is not in list
>>>
```

CHAPTER 9

SETS IN PYTHON

The data type "set", which is a so-called "collection" type, has been in Python since version 2.4. contain. A set contains an un ordered collection of unique and immutable elements. In other words, an element cannot appear more than once in a set object, which is possible with lists and tuples. The data type set is the Python implementation of sets as they are known from mathematics.

CREATE SETS

If you want to create a lot, this is very easy in Python3. The usual mathematical notation is used.

```
>>>staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
>>> print (cities)
{'Munich', 'Berlin', 'Frankfurt', 'Hamburg'}
>>>'Berlin' in cities
True
>>>'Cologne' in cities
False
```

If you look closely at the example of a set above, you will surely be surprised that the order of the cities is printed differently from print than the cities in the definition of the set. This is because sets in general, and also in the Python programming language, contain an unordered collection of different objects. So there is no order for quantities, and therefore the print output can be arbitrary. You can find more on this in the last section of this chapter.
In Python, however, the elements can only be immutable types, ie

they cannot be lists, for example. We demonstrate this in the following example, in which we try to create a set that should contain lists of two with cities and their population (as of 2013):

```
>>>staedte = {['Berlin', 3.42], ['Hamburg', 1.75], ['München', 1.41], ['Köln', 1.03], ['Frankfurt', 0.70], ['Stuttgart' , 0.60]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

If you use tuples (unchangeable) instead of lists (changeable) as elements, then it works:

```
>>>staedte = {('Frankfurt', 0.7), ('Stuttgart', 0.6), ('Hamburg', 1.75), ('Berlin', 3.42), ('Köln', 1.03), ('München' , 1.41)}
```

One more note: Those who are already familiar with dictionaries or who have already worked through our chapter on dictionaries may be surprised that we also use the braces for sets.
In addition to the "set" data type, there is also a "set ()" function. With the function set () you can convert sequential data types, e.g. lists, into sets. In the following small example, we define a list of words that we convert into a set using the set () function:

```
>>>liste_von_woertern = ["good", "helpful", "better", "optimal"]
>>>quantity_of_woertern = set (list_of_woertern)
>>>crowd_of_worthers
{'helpful', 'better', 'optimal', 'good'}
>>>
```

The set function is often used to remove multiple occurrences from lists or other sequential data types. We want to show this in an example with a list of cities from Switzerland, Austria and Germany. The city of Zurich appears twice in this list. As expected, it only appears once in the generated quantity, since there cannot be multiple occurrences in a quantity:

```
>>>liste_von_staedten = ["Frankfurt", "Zürich", "Bern", "Stuttgart", "Freiburg", "Ulm", "Hamburg", "Munich", "Nürnberg", "Zürich", "Bregenz", "Salzburg", "Vienna"]
```

```
>>>list_of_states
```

```
['Frankfurt', 'Zürich', 'Bern', 'Stuttgart', 'Freiburg', 'Ulm', 'Hamburg', 'München', 'Nürnberg', 'Zürich', 'Bregenz', 'Salzburg', ' Vienna']
```

```
>>>quantity_of_states = set (list_of_states)
```

```
>>>crowd_of_states
```

```
{'Vienna', 'Bern', 'Hamburg', 'Nuremberg', 'Frankfurt', 'Bregenz', 'Zurich', 'Munich', 'Ulm', 'Stuttgart', 'Freiburg', 'Salzburg'}
```

```
>>>
```

You can then convert the quantity back into a list if you need a list as a data type:

```
>>>liste_von_staedten = list (quantity_of_staedten)
```

```
>>>list_of_states
```

```
['Vienna', 'Bern', 'Hamburg', 'Nuremberg', 'Frankfurt', 'Bregenz', 'Zurich', 'Munich', 'Ulm', 'Stuttgart', 'Freiburg', 'Salzburg']
```

```
>>>
```

As we can see, this approach has a minor flaw. Multiple occurrences are no longer included in the results list, but the original order is usually no longer available.
We now come to another application of the set function. In the following example, we use them to separate a string into its characters:

```
>>> x = set ("A good Python tutorial")
```

```
>>> x
```

```
{'t', 'y', 'h', 'o', 's',' P ',' a ',' l ',' e ',' E ',' r ',' n ',' - ','i',' g ',' u ',' T ',''}
```

```
>>> type (x)
```

```
<class 'set'>
```

```
>>>
```

SETS OF IMMUTABLE ELEMENTS

Sets are implemented in such a way that they do not allow mutable objects. The following example demonstrates that we cannot have lists as elements, for example:

```
>>> cities = set ((("Python", "Perl"), ("Paris", "Berlin", "London")))
>>> cities = set ((["Python", "Perl"], ["Paris", "Berlin", "London"]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

FROZEN SETS

Even if sets cannot contain changeable elements, they are changeable themselves. For example, we can add new elements:

```
>>> cities = {"Frankfurt", "Basel", "Freiburg"}
>>>cities.add ("Strasbourg")
>>> cities
{'Freiburg', 'Frankfurt', 'Basel', 'Strasbourg'}
```

Frozen sets are like sets, but they cannot be changed. So they are immutable:

```
>>> cities = frozenset (["Frankfurt", "Basel", "Freiburg"])
>>>cities.add ("Strasbourg")
Traceback (most recent call last):
  File "<stdin & module>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

```
>>>
```

OPERATIONS ON "SET" OBJECTS

add method

With the add method, an object is added to a set as a new element if it does not already exist. It should be noted that this is an unchangeable element.
In the following example, we insert a string:

```
>>> colors = {"red", "green"}
>>>colours.add ("yellow")
>>> colors
{'green', 'yellow', 'red'}
>>>colours.add (["black", "white"])
Traceback (most recent call last):
  File "<stdin & module>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

Of course, an object is only inserted as a new element if it is not yet included. If it is already included, calling the method has no effect.

clear method

All elements of a set are removed. The quantity is then empty, as we can see in the following example:

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

copy method

copy creates a flat copy of a quantity that is returned. We demonstrate the use of an example:

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

Only for those who believe that a simple assignment could also be enough:

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

The assignment "cities_backup = more_cities" creates only one pointer, ie another name for the same object.

difference method

This method returns the difference between two or more quantities. As always, we illustrate this with an example:

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

```
{'e', 'a'}
>>>
```

Instead of using the "difference" method, we could have used the "-" operator:

```
>>> x - y
{'e', 'd', 'a'}
>>> x - y - z
{'e', 'a'}
>>>
```

difference_update method

The method "difference_update" removes all elements of a different set from a set. "x.difference_update (y)" is synonymous with "x = x - y"

```
>>> 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
{'e', 'd', 'a'}
>>>
```

discard method

When discard (el) is called, the el element is removed from a set if it is included. If el is not included in the set, nothing happens.

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

remove method

The "remove" method works like discard (), but if el is not included in the set, an error is generated, ie a KeyError:

```
>>> x = {"a", "b", "c", "d", "e"}
>>>x.remove ("a")
>>> x
{'e', 'c', 'b', 'd'}
>>>x.remove ("z")
Traceback (most recent call last):
  File "<stdin & module>", line 1, in <module>
KeyError: 'z'
>>>
```

union method

The "union" method returns the union of two sets as a new set, ie all elements that occur in both sets.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d", "e", "f", "g"}
>>>x.union (y)
{'d', 'a', 'g', 'c', 'f', 'b', 'e'}
```

To form the unions you can also use the pipe character "|" use as operator:

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d", "e", "f", "g"}
>>> x | y
{'d', 'a', 'g', 'c', 'f', 'b', 'e'}
>>>
```

intersection method

The intersection method can be used to intersect two sets, as we can see in the following example.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d", "e", "f", "g"}
>>>x.intersection (y)
{'e', 'c', 'd'}
```

This can also be formulated with the "&" sign:

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d", "e", "f", "g"}
>>> x & y
{'e', 'c', 'd'}
>>>
```

isdisjoint method

This method returns True if two sets have an empty intersection.

```
>>> x = {"a", "b", "c"}
>>> y = {"c", "d", "e"}
>>>x.isdisjoint (y)
False
```

```
>>>
>>> x = {"a", "b", "c"}
>>> y = {"d", "e", "f"}
>>>x.isdisjoint (y)
True
>>>
```

issubset method

x.issubset (y) returns True if x is a subset of y. "<=" can be used instead of calling the method. "<" checks whether it is a real subset: If x <y, then y contains at least one element that is not contained in x.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d"}
>>>x.issubset (y)
False
>>>y.issubset (x)
True
>>> x <y
False
>>> y <x # y is a real subset of x
True
>>> x <x # a set can never be a real subset of itself.
False
>>> x <= x
True
>>>
```

issuperset method

x.issuperset (y) returns True if x is a superset of y. "> =" can be used instead of calling the method. ">" checks whether it is a real superset: If x> y, then x contains at least one element that is not contained in y.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d"}
>>>x.issuperset (y)
True
>>> x> y
True
>>> x> = y
True
>>> x> = x
True
>>> x> x
False
>>>x.issuperset (x)
True
>>>
```

pop

pop () returns any element of the set. This element is removed from the set. The method generates a KeyError if the quantity is empty.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> x
{'e', 'c', 'b', 'd', 'a'}
>>>x.pop ()
'e'
```

```
>>>x.pop ()
'c'
>>>x.pop ()
'b'
>>>x.pop ()
'd'
>>>x.pop ()
'a'
>>>
```

Even if "pop" returns the elements in the order of the internal representation of the set, the phrase "pop () returns any element of the set." The amount we have in the "head" is in the example above {"a", "b", "c", "d", "e"} and the order in which pop () returns the elements is' e ',' c ',' b ',' d ',' a '.
That this order is not always the same for this set can be seen in the following new interactive Python session:

```
>>> x = {"a", "b", "c", "d", "e"}
>>>x.pop ()
'a'
>>>x.pop ()
'b'
>>>x.pop ()
'e'
>>>x.pop ()
'd'
>>>x.pop ()
'c'
>>>
```

ANY OR RANDOM ORDER

We learned that there is no set order for the elements in sets, and that you cannot define such an order. As we said at the beginning, the order of quantities is arbitrary. "Any" should not be confused with "random". In principle, it could be the case, for example, that Python always saves the elements of a certain set internally in the same order, which can then be seen when the set is output. But this is not the case. The order depends on a hash, a special function used in dictionaries and sets. With different program runs this can lead to different representations of the crowd. We demonstrate this in the following example:

```
bernd @ saturn: ~ $ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = {"a", "b", "c", "d", "e"}
>>> x
{'d', 'a', 'e',   'c', 'b'}
>>>
bernd @ saturn: ~ $ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
>>> x = {"a", "b", "c", "d", "e"}
>>> x
{'b', 'e',   'c', 'a', 'd'}
```

```
>>>
bernd @ saturn: ~ $ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>staedte = ['Hamburg', 'Munich', 'Frankfurt', 'Berlin']
>>> x = {"a", "b", "c", "d", "e"}
>>> x
{'b', 'a', 'c', 'e',   'd'}
>>>
>>> print (x)
{'b', 'a', 'c', 'e',   'd'}
>>>
```

CHAPTER 10

FUNCTIONS AND MODULES

The concept of a function is probably one of the most important in mathematics. Functions are also often used in programming languages to implement mathematical functions in algorithms. Such a function calculates one or more values and is completely dependent on the transferred input values, the so-called parameters.

Generally speaking, a function is a structuring element in a programming language to group a set of instructions so that they can be used multiple times in the program. Without functions, you could only copy code to reuse it. Then you would have to adapt the code accordingly. Later changes would then have to be made at all copied points, which would of course be very error-prone. The use of functions increases the comprehensibility and quality of a program or script. In addition, using functions also lowers software development and maintenance costs.Functions are known by different names in different programming languages. So they are also known as subroutines, routines, procedures, methods and subroutines.

PYTHON SYNTAX

Functions are initiated with the keyword def

```
def function-name (parameter list):
   Instructions)
```

The parameter list consists of one or more identifiers, separated by commas. The parameters of the definition are called arguments when the function is called. Most of the time, however, the two terms are incorrectly used like synonyms.
Parameters can be mandatory and optional. The optional

parameters (0 or more) follow the mandatory ones.
The function body, i.e. the instructions that are executed when the function is called, is marked in Python by a homogeneous indentation. So just like all other blocks in Python. The function definition ends when an instruction is issued that is again at the same indentation level as the head of the function.
The function body can contain one or more return statements. These can be located anywhere in the functional body. A return statement ends the function call and the result of the expression that follows the return statement is returned to the calling point. If no expression follows the return, the special value None is returned. If the end of a function body is reached without encountering a return statement, the function call ends and the value None is also returned.
Example:

```
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))
```

The output of the above script with the fahrenheit function:

```
22.6: 72.68
25.8: 78.44
27.3: 81.14
29.8: 85.64
```

EXAMPLE OF A FUNCTION WITH OPTIONAL PARAMETERS

Functions can also have optional parameters. They are also called default parameters. These are parameters that do not have to be specified when called. In this case, default values are used for these

parameters.
We show this with an example. The following little script, which is not very useful, greets a person by name. However, if no name is passed when the call is made, it only prints "Hello everybody!":

```
def Hello (name = "everybody"):
"""Greets a person"""
   print ("Hello" + name + "!")
Hello ("Peter")
Hello()
```

The output then looks like this:

```
Hello Peter!
Hello everybody!
```

DOCSTRING

The first statement of a function body is usually a character string that can be queried as a function name .__ doc__.
This instruction is called **docstring** .
Example:

```
def Hello (name = "everybody"):
"""Greets a person"""
   print ("Hello" + name + "!")
print ("The docstring of the function Hello:" + Hello .__ doc__)
```

The edition:

```
The docstring of the function Hello: Greets a person
```

KEYWORD PARAMETERS

This is an alternative way of calling a function. The function definition itself does not change.
An example:

```
def sumsub (a, b, c = 0, d = 0):
    return a - b + c - d
print (sumsub (12.4))
print (sumsub (42.15, d = 10))
```

Only keywords that have not already been used as position arguments may be used as keyword parameters. We can see the advantage in the example. If we had no keyword parameters, we would have to give all arguments in the second function call, although c only has to have the default value:

```
print (sumsub (42,15,0,10))
```

RETURN VALUES

In our previous examples, we used return statements in the sumsub function but not in Hello. So you can see that it is not absolutely necessary to have a return statement in a function. But what will be returned if we don't have an explicit return instruction. Let's take a look at an example:

```
def no_return (x, y):
     c = x + y
res = no_return (4.5)
print (res)
```

When we start the small script above, None is printed, ie the special value None is returned by the no_return function. This shows that a function that ends without a return statement returns None. However, this value is also returned if a return statement ends a function without a subsequent expression, as we can see in the following example:

```
def empty_return (x, y):
    c = x + y
    return
res = empty_return (4,5)
print (res)
```

Otherwise, the value of the expression is returned after the return. In the next example 9 is returned:

```
def return_sum (x, y):
    c = x + y
    return c
res = return_sum (4,5)
print (res)
```

MULTIPLE RETURN VALUES

A function can return exactly one value, or we should better say exactly one object. For example, an object can be a numerical value, such as an integer or a decimal number (float), but also a complex object, such as a list or a dictionary. For example, if we want to return three integers, we have the option of packing them into a tuple or a list and passing this list or tuple object as a return value. This means that we can indirectly return any number of values in a very simple way.
In the following example, the function fib_interval calculates the Fibonacci limit for any positive number, that is, it returns a 2-tuple. The first element is the largest Fibonacci number that is less than or equal to x and the second component is the smallest Fibonacci number that is greater than or equal to x. The return value is saved directly to the variables lub and sup using "unpacking":

```
def fib_intervall (x):
"""returns the largest fibonacci
```

```
    number smaller than x and the lowest
fibonacci number higher than x """
    if x <0:
        return -1
    (old, new, lub) = (0,1,0)
    while True:
        if new <x:
lub = new
            (old, new) = (new, old + new)
        else:
            return (lub, new)

while True:
    x = int (input ("Your number:"))
    if x <= 0:
        break
    (lub, sup) = fib_intervall (x)
    print ("Largest Fibonacci Number smaller than x:" + str (lub))
    print ("Smallest Fibonacci Number larger than x:" + str (sup))
```

CHAPTER 11

CLASS AND TYPES

In this chapter, we'd like to introduce you to the magical behavior of Python when we define a class or create an instance of a class. You may be wondering, "Do I really need to know the details?" Probably not ... or you are one of those who create classes at an advanced level.
First we focus on the connection between "type" and "class". When you defined a class, you may have wondered what happens after the "lines". We have already seen that type (object) returns the class of the instance:

```
x = [4, 5, 9]
y = "Hello"
print (type (x), type (y))
```

We get the following output:

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

If type () is applied to the names of the classes themselves, we get "type" as a result.

```
print (type (list), type (str))
Output:
<class 'type'><class 'type'>
```

We get the same result with the following code:

```
x = [4, 5, 9]
y = "Hello"
print (type (x), type (y))
print (type (type (x)), type (type (y)))
```

```
Output:
<class 'list'><class 'str'>
<class 'type'><class 'type'>
```

This shows us that classes like 'list' and 'int' are instances of the class 'type'. In general, it can be said that all of the classes we have created so far are instances of the "type" class.
In Python3 there is no difference between "classes" and "types". In most cases they are used interchangeably.
The fact that classes are instances of the class "type" allows us to program meta classes. We can create classes that inherit from the "type" class. A meta class is therefore a subclass of the class "type".
Instead of a single argument, type can be called with three parameters:

```
type (classname, superclasses, attributes_dict)
```

If type is called with three arguments, it returns a new type object. This offers us a dynamic form of the class statement.

1. "classname" is the string that specifies the class name and becomes the name attribute
2. "superclasses" is a list or a tuple with the upper classes of our classes. The list or tuple becomes the bases attribute.
3. "attributes_dict" is a dictionary that acts as the namespace of our class. It contains the definitions of the class body and becomes the dict attribute.

Let's look at a simple class definition:

```
class A:
   passport
x = A ()
```

```
print (type (x))
```

We get the following output:

```
<class '__main __. A'>
```

We can also use type () for the class definition described above:

```
A = type ("A", (), {})
x = A ()
print (type (x))
```

We also get the following output:

```
<class '__main __. A'>
```

Generally this means that we have classes with

```
type (classname, superclasses, attributedict)
```

can define.
When we call type (), the **call** method of type is called. The **call** method calls 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. Then the **init** method initializes the created object.

```
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__)
```

After executing the code, we get the following output:

```
Marvin
Hi, I am Marvin
Marvin
Hi, I am Marvin
{'name': 'Marvin'}
{'name': 'Marvin'}
```

The class definitions of Robot and Robot2 are completely different syntactically. But logically speaking, both implement exactly the same thing.What Python does in the first example is that all methods and attributes of the Robot class are collected and then passed as parameters to the attributes_dict argument when the type is called. So, Python does the same thing we did with the Robot2 class.

OBJECT ORIENTED PROGRAMING

Even if Python is an object-oriented programming language without any ifs and buts, in the previous chapters we only dealt indirectly with object-oriented programming (OOP). With Python, small scripts or programs can be written easily and efficiently, even without being modeled in an object-oriented manner. Experience has shown that total programming beginners find it easier if they are not immediately confronted with all the principles of OOP. You have enough problems to understand assignments, conditional statements or loops and above all to use them correctly. But in many situations, the OOP represents a significant qualitative improvement in the implementation of a problem. But even if we have avoided object-oriented programming in the previous chapters, so it was mostly present in our exercises and examples. We used class objects and methods without actually knowing their existence. In this chapter, we now give a basic introduction to Python's object-oriented approach. OOP is one of the most powerful programming options in Python, but, as we have seen, you do not have to use it, which means that you can also write extensive and efficient programs without using OOP techniques.Even though many programmers and computer scientists consider the OOP to be a modern achievement, its roots go back to the 1960s. The first programming language to use objects was "Simula 67" by Ole-Johan Dahl and Kristen Nygard. As the name suggests, this language was introduced in 1967.

OBJECTS, INSTANCES AND CLASSES

A basic concept of object-oriented programming is to combine data and their functions (methods), ie functions that can be applied to this data, in one object and encapsulate them outside, so that the users of the classes and also methods of external objects can use them Cannot manipulate data.Objects are defined via classes. Classes are templates - one could also say "blueprints" - according to which objects - which in this context are also referred to as instances - are created during the run time of the program. A class is a formal description of how an object is made, ie which attributes and which methods it has. A class should not be confused with an object. Instead of an object, one also speaks of an instance of a class, ie the terms "instance" and "object" are mostly used synchronously.You can also see a class figuratively like a cooking or baking recipe. For example, consider the recipe for a strawberry cake. In principle, such a recipe can be seen as a class. That is, the recipe determines what an instance of the class should look like. If someone bakes a cake according to this recipe, he creates an instance or an object of this class. There are then various methods of processing or changing this cake, such as `` mixing dough ''. A strawberry cake belongs to a superordinate class "cake", which inherits its properties, e.g. that a cake can be used as a dessert, in sub-classes such as strawberry cake, sponge cake, tarts and so on.

In the OOP, an object denotes the mapping of a real object with its properties and behavior (methods) into a program. An object can always be described by two things:

1. what it can do or what we can do with it in a program
2. what we know about it.

ENCAPSULATION OF DATA

Another key advantage of OOP is the encapsulation of data. Properties can only be accessed using access methods. These methods can include plausibility tests, data type conversions or any kind of calculations, and they (or "only" them) have "information" about the actual implementation. Robot and account In the next section we will write a robot class in Python. This will contain information about the year of construction and the name of a robot, for example. This information is also referred to as the properties or attributes of an instance. For example, it is useful to save the name of a robot as a string in an attribute. Encapsulation now means that we cannot access this string directly. For example, we need to call a method called "GetNamen ()" to get the name of a robot. The principle of data encapsulation can also be seen in the model of the account class. The method of setting the date of birth can, for example, check whether the date is correct: This is how you can intercept if someone enters a date in the future due to a typing error. You could also generally check whether the information is within a certain range. For example, a current account for children under 14 should not be possible or investments by new customers over 100 years old are considered extremely unlikely.

INHERITANCE

Based on a general robot class that only knows a name and a year of construction, we could define other robot classes such as "industrial robots" that can be used stationary on an assembly line, moving robots with wheels, legs or caterpillars and so on. Each of these classes then inherits the possibility to have a name and a year of construction from the base class.

OBJECTS AND INSTANCES OF A CLASS

One of the many classes built into Python is the list class, which we have used many times in our exercises and examples. The list class provides a wealth of methods with which we can, for example, build lists, view, change and remove elements:

```
>>> 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 designate two instances of the list class. So far we have simply said that x and y are lists. In the following we will use the terms "object" and "instance" Synchronously, as is also the case in other introductions.

ENCAPSULATION OF DATA AND METHODS

pop and append from the example above are methods of the list class. pop returns the "top" or the element with the highest index in the list and removes this element. However, we do not know how the lists are stored internally in memory. We don't need this information either, because the list class provides us with methods to access the stored data "indirectly". Methods are of particular importance in connection with data encapsulation. We'll look at encapsulation in more detail later. It prevents direct access to the internal data structure.

A MINIMAL CLASS IN PYTHON

In the following, we will demonstrate the most important terms of object-oriented programming and their implementation in Python using an example "robot class". We start with a simple class in Python that we call "robots".

```
class robot:
   passport
```

This example shows the basic syntactic structure of a class: A class consists of two parts: the head and the body. The header usually consists of only one line: the keyword class, followed by a space, any name, - in our case robot - a comma-separated list of super

classes in brackets and a colon as the last character. If there are no super classes, the super classes and parentheses are omitted. In principle, there can also be an empty pair of brackets in front of the colon. You do not need to understand the pair of parentheses with the list of upper classes at this point in time, as we will only discuss them later!
The body of a class consists of an indented sequence of instructions which, as in our example, can consist of only one pass instruction.
We have already defined a simple Python class called robot. We can also use this class:

```
class robot:
    passport
if __name__ == "__main__":
    x = robot ()
    y = robot ()
    y2 = y
    print (y == y2)
    print (y == x)
```

In the example above, we created two different robots x and y. We also created an alias y2 for y with y2 = y. This is just another name for the same object, that is, a reference. Therefore, the program also delivers

```
True
False
```

Explanation on __name__ : In the above program we used the variable __name__. Each Python module has a name defined in the built-in attribute __name__. Let us assume that we have saved the above program as a module with the name "robots" under "robots.py". If this module is imported with "import robots", the built-in attribute __name__ has the value "robots". However, if the robots.py file is called as an independent program, ie using "$ python3 robots.py", then this variable has the value '__main__'.

PROPERTIES AND ATTRIBUTES

Our robots have no properties. Not even a name, as is common for "decent" robots. A type designation, year of construction and so on would be conceivable as further properties. Properties are called attributes in object-oriented programming.Any attribute name can be assigned to an instance. They are connected with a dot to the name of the instance. In the following, we dynamically create attributes for the robot name and the robot year. Please note that this has nothing to do with the actual attributes as we will use them in classes:

```
>>> class robot:
... passport
...
>>> x = robot ()
>>> y = robot ()
>>>
>>> x.name = "Marvin"
>>> year of construction = 1979
>>> y.name = "Caliban"
>>> year of construction = 1993
>>> print (x.name)
Marvin
>>>
```

Incidentally, attributes can also be assigned to the class object itself or - regardless of the OOP functions - as we see below:

```
>>> class robot:
```

```
... passport
...
>>>Robot.number = 1000
>>> print (robot.number)
1000
>>> def f (x):
... return 42
...
>>>f.color = "red" # whatever it means to assign a color attribute
to a function :-)
>>> print (f.color)
red
>>>
>>> # our function is not affected:
...
>>> f (10)
42
>>>
```

Function attributes can be used, for example, to replace static function variables, such as some you know from C, C ++ or Java. Python has no static function variables!
In the following function, the "counter" attribute is used to count the number of times the function is called:

```
def f (x):
   if hasattr (f, "counter"): # Alternative: if "counter" in you (f):
f.counter + = 1
    else:
```

```
f.counter = 0
    return x + 3

for i in range (10):
    f (i)

print (f.counter)
```

One more detail, which is not so important at the moment. So you are welcome to continue with the next subsection "Methods".
The objects of most classes have an attribute dictionary __dict__, in which the attributes are stored with their values, as we see in the next example.

```
>>> class Robot:
... passport
...
>>> x = Robot ()
>>> x.name = "Marvin"
>>>x.age = 5
>>>
>>>x .__ dict__
{'name': 'Marvin', 'age': 5}
>>>
```

The dynamic creation of attributes for instances is seen by some as a blessing and others as a curse. But attributes like those used in object-oriented programming are not created in this way.
If we want to create instance attributes for our robot class, we have to do this directly in the class definition. Instance attributes are the properties that describe the individual instances, meaning that our

robots generally have different names, and certainly a different serial number. We need methods to create instance attributes in a class definition.

METHODS

In the following we want to show how to define methods in a class. We will add a SageHallo method to our empty robot class. A method differs externally from a function only in two aspects:

- It is a function that is defined within a class definition.
- The first parameter of a method is always a reference to the instance from which it is called. This reference is usually called "self".

Note: In principle, you could choose any name instead of "self", including "this", which Java or C ++ programmers might like more. "self" is just a convention.
We are now expanding our robot class with a SageHallo method that simply writes "Hello" when it is called:

```
class robot:
    def SageHallo (self):
        print ("hello")
if __name__ == "__main__":
    x = robot ()
x.SageHallo ()
```

We see in the code that the parameter "self" only appears when a method is defined. It is not specified when called. In comparison to function calls, this is strange at first, ie we define a method with a parameter "self" and call it up apparently without parameters. But if we take a closer look at the call, we see that we are not just calling SageHallo (), but that the instance x appears before the point. Herein lies the secret: It is as if we had called SageHallo (x). In other words, we pass a reference to the instance x to self. For further understanding: One should actually call the method of a class via the class name. In this case, the instance is passed as an argument, i.e. Robot.SageHallo (x).

INSTANCE VARIABLES

We now want to change our SageHallo method so that it says "Hello, my name is Marvin" when the robot is called "Marvin". This brings us back to the instance attributes. Because an instance must be able to remember its name. In our initial example, we defined attributes outside of the class definition with instructions of the type x.name = "Marvin", x.baujahr = 1979 and y.name = "Caliban". In a way, we can do the same thing within a class definition in the methods. However, of course we have no idea of the instance name, i.e. x or y. However, we do not need to know this name, since the formal parameter self is a reference to the current instance. With this knowledge, we are now writing further functions

```
class robot:
    def SageHallo (self):
        print ("Hello, my name is" + self.name)
    def SetNames (self, name):
        self.name = name
    Def Set year of construction (self, year of construction):
self.baujahr = year of construction


if __name__ == "__main__":
    x = robot ()
x.SetzeNamen ("Marvin")
x.SetzeBaujahr (1979)
    y = robot ()
```

```
y.SetNames ("Caliban")
y.SetzeBaujahr (1993)
x.SageHallo ()
y.SageHallo ()
```

There is the following to say about the SetNames method, which also answers two frequently asked questions:

- Yes, you can use any name instead of self, including this one. However, you shouldn't do this because it violates a Python convention. In addition, other programmers may have difficulty understanding the code. Some development environments, such as eclipse, also give a warning if you do not adhere to this convention.
- The attribute name does not have to be the same as the name of the formal parameter. However, it often has the same name. We could also have called the parameter, for example, "n".

So the following Python code is also executable, but by no means recommended:

```
class robot:
     def SageHallo (self):
        print ("Hello, my name is" + self.name)
    def SetNames (this, n):
        this.name = n
    Def Set year of construction (self, year of construction):
self.baujahr = year of construction



if __name__ == "__main__":
    x = robot ()
x.SetzeNamen ("Marvin")
x.SetzeBaujahr (1979)
    y = robot ()
y.SetNames ("Caliban")
y.SetzeBaujahr (1993)
x.SageHallo ()
y.SageHallo ()
```

What is particularly ugly about the code above is that we used this once and self once, so that we were not consistent in our naming.There is also something to say about style: According to PEP8, the official "Style Guide", the following applies: "Method definitions within a class are separated by a single blank line." a class are separated by a single blank line. ") However, this is not adhered to in many cases. Even the official Python documentation at python.org is not unique here! We will not always adhere to this convention below, above all to save space.Butbesides style and naming for the first parameter, there is a much more serious problem. When we create a new robot, we have to follow three instructions each time. In our example these are, instantiation using x = robot (), naming x.SetzeNamen ("Marvin") and setting the year of construction x.SetzeBaujahr (1979). This procedure is cumbersome, prone to errors and, above all, it does not correspond to the usual procedure in the OOP.

The __init __ method

We want to define the attributes immediately after creating an instance. __init__ is a method that is called immediately and automatically after the creation of an instance. This name is fixed and cannot be chosen freely! __init__ is one of the so-called magic methods, of which we will learn more in the following chapters. The __init __ method is used to initialize an instance. Python does not have an explicit constructor or destructor as you know it in Java or C

++. The actual constructor is implicitly started by Python and __init__ serves, as the name suggests, to initialize the attributes. However, the __init __ method is started immediately after the actual constructor, and this creates the impression that it is a constructor.

```
>>> class A:
... def __init __ (self):
... print ("__ init__ was executed!")
...
>>> x = A ()
__init__ has been executed!
>>>
```

We are now writing an initialization method __init__ for our example class in order to be able to set or transfer the name and the year of manufacture during instantiation, ie x = robot ("Marvin", 1979).

```
class robot:
    def __init __ (self, name, year of construction):
        self.name = name
self.baujahr = year of construction
    def SageHallo (self):
        print ("Hello, my name is" + self.name)
    def NewName (self, name):
        self.name = name
    def new year of construction (self, year of construction):
self.baujahr = year of construction


if __name__ == "__main__":
  x = robot ("Marvin", 1979)
  y = robot ("Caliban", 1993)
x.SageHallo ()
y.SageHallo ()
```

If you write x = robot ("Marvin", 1979), then logically speaking it behaves as if you would instantiate an instance x first, i.e. x = robot (), - which of course no longer works because of our __init __ method, which requires two arguments when calling! -, and then call the __init __ method with x .__ init __ ("Marvin", 1979). (However, the __init __ method cannot be called from outside, since it begins with a double underscore and is therefore a private method.)Using the class now looks much more "tidy" and clearer, but we are hurting nor the principle of data encapsulation or data abstraction, which we will discuss below.
In the following we want to demonstrate an error message that often raises questions for beginners. We try to create a robot in the following interactive Python shell. Instead of the correct instantiation x = robot ("Marvin", 1979) we call robots without parameters:

```
>>> from robots2 import robots
>>> x = robot ()
Traceback (most recent call last):
  File "", line 1, in
TypeError: __init __ () takes exactly 3 arguments (1 given)
>>>
```

We know that we have to give a name and a year of construction when defining a new robot. At first glance, the error message "__init __ () takes exactly 3 arguments (1 given)" seems surprising. We hadn't given one argument and we should have given two? If we look at the exact definition of __init __ (), the apparent contradiction clears up immediately, because this method really has three parameters: self, name, year of construction. "self" is generated implicitly, hence the message "1 given".