

HISTORY OF PYTHON

EASY AS ABC

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

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

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



COMEDY, SNAKE OR PROGRAMMING LANGUAGE

So, what about the name "Python": Most people think about snakes, and even the logo depicts two snakes, but the origin of the name has its root in British humour. Guido van Rossum, the creator of Python, wrote in 1996 about the origin of the name of his programming language¹: "Over six years ago, in December 1989, I was looking for a 'hobby' programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been

thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

THE ZEN OF PYTHON

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

DEVELOPMENT STEPS OF PYTHON

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

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

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

Python flourished for another 8 years in the versions 2.x before the next major release as

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

Some changes in Python 3.0:

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

¹ January 13, 2003, <http://www.artima.com/intv/pythonP.html> ² Foreword for "Programming Python" (1st ed.) by Mark Lutz, O'Reilly

THE INTERPRETER, AN INTERACTIVE SHELL

THE TERMS 'INTERACTIVE' AND 'SHELL'



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

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

The interactive shell is also interactive in the way that it stands between the commands or actions and their execution. This means the Shell waits for commands from the user, which it executes and returns the result of the execution. After this the shell waits for the next input.

A shell in biology is a calcium carbonate "wall" which protects snails or mussels from its

environment or its enemies. Similarly, a shell in operating systems lies between the kernel of the operating system and the user. It's a "protection" in both direction. The user doesn't have to use the complicated basic functions of the OS but is capable of using simple and easier to understand shell commands. The kernel is protected from unintended incorrect usages of system function.

Python offers a comfortable command line interface with the Python shell, which is also known as the "Python interactive shell".

It looks like the term "interactive Shell" is a tautology, because "Shell" is interactive on its own, at least the kind of shells we have described in the previous paragraphs.

USING THE PYTHON INTERACTIVE SHELL

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

```
python
```

Python comes back with the following information:

```
$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

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

```
bernd@venus:~$ $ python
python      python2.7    python3.5    python3m
python2     python3      python3.5m
bernd@venus:~$ python
```

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

```
$ python3
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
```

```
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Once the Python interpreter is started, you can issue any command at the command prompt "`>>>`".

Let's see, what happens, if we type in the word "hello":

```
>>> hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
>>>
```

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

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

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

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

```
>>> "Hello World"
'Hello World'
>>> 3
3
>>>
```

HOW TO QUIT THE PYTHON SHELL

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

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

THE SHELL AS A SIMPLE CALCULATOR

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

```
>>> 4.567 * 8.323 * 17  
646.1893969999999  
>>>
```

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

1. exponents and roots
2. multiplication and division
3. addition and subtraction

This means that we don't need parenthesis in the expression "3 + (2 * 4):

```
>>> 3 + 2 * 4  
11  
>>>
```

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

```
>>> _  
11  
>>>
```

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

```
>>> _ * 3  
33  
>>>
```

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

USING VARIABLES

It's simple to use variables in the Python shell. If you are an absolute beginner and if you don't know anything about variable, please confer our chapter about variables and data types.

Values can be saved in variables. Variable names don't require any special tagging, like they do in Perl, where you have to use dollar signs, percentage signs and at signs to tag variables:

```
>>> maximal = 124
>>> width = 94
>>> print(maximal - width)
30
>>>
```

MULTILINE STATEMENTS

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

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

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

After having input "for character in l:" the interpreter expects the input of the next line to be indented. In other words: The interpreter expects an indented block, which is the body of the for loop. This indented block will be iterated. The interpreter shows this "expectation" by showing three dots "..." instead of the standard Python interactive prompt ">>>".

Another special feature of the interactive shell: When we have finished with the indented lines, i.e. the block, we have to enter an empty line to indicate that the block is finished.

Attention: The additional empty line is only necessary in the interactive shell! In a Python program, it is enough to return to the indentation level of the "for" line, the one with the colon ":" at the end.

STRINGS

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

```
>>> "Hello" + " " + "World"
'Hello World'
```

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

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

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

```
>>>
```

There is a multiplication on strings defined, which is essentially a multiple concatenation:

```
>>> ".-." * 4  
'.-.-.-.-.'  
>>>
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

EXECUTE A PYTHON SCRIPT

So far we have played around with Python commands in the Python shell. We want to write now our first serious Python program. You will hardly find any beginner's textbook on programming, which don't start with the "nearly mandatory" "Hello World" program, i.e. a program which prints the string "Hello World". This looks on the Python shell like this:

```
$ 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.
>>> print("Hello World!")
Hello World!
>>>
```

But, as we said at the beginning, we want to write a "serious" script now. We use a slight variation of the "Hello World" theme. We have to include our print statement into a file. To save and edit our program in a file we need an editor. There are lots of editors, but you should choose one, which supports syntax highlighting and indentation. Under Linux you can use vi, vim, emacs, geany, gedit and umpteen others. The emacs works under windows as well, but notepad++ may be the better choice in many cases.

So, after you have found the editor of your choice, you can input your mini script, i.e.

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

and save it as **my_first_simple_script.py**.

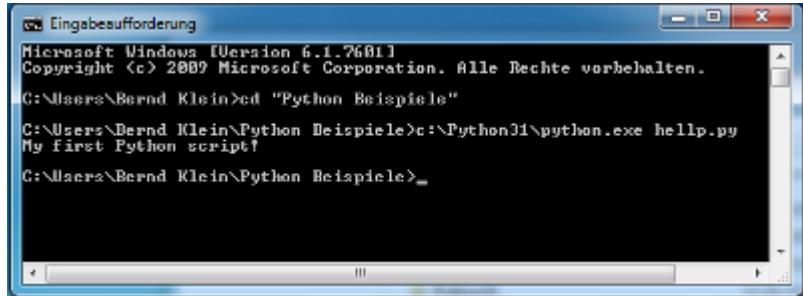
The suffix .py is not really necessary under Linux but it's good style to use it. But the extension is essential, if you want to write modules.

START A PYTHON SCRIPT

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

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

It can be started under Windows in a Command prompt (start -> All Programs -> Accessories -> Command Prompt):



PYTHON INTERNALS

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

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

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

```
>>> import py_compile
>>> py_compile.compile('my_first_simple_script.py')
>>>
```

or using the following command at the shell prompt

```
python -m py_compile my_first_simple_script.py
```

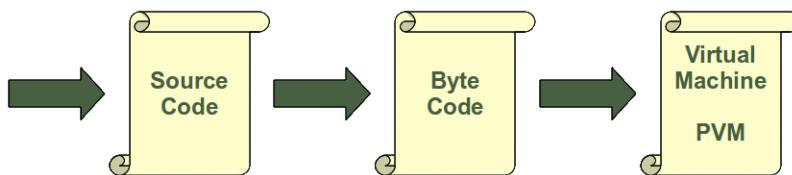
Either way, you may notice two things: First, there will be a new subdirectory "`__pycache__`", if it hasn't already existed. You will find a file "`my_first_simple_script.cpython-34.pyc`" in this subdirectory. This is the compiled version of our file in byte code.

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

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

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

Whenever a Python program is called, Python will check, if there exists a compiled version with the `.pyc` suffix. This file has to be newer than the file with the `.py` suffix. If such a file exists, Python will load the byte code, which will speed up the start up time of the script. If there exists no byte code version, Python will create the byte code before it starts the execution of the program. Execution of a Python program means execution of the byte code on the Python Virtual Machine (PVM).



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

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

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

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

```
My first simple Python script!
>>> exit()
monty@python:~/tmp$ ls
my_first_simple_script.py  my_first_simple_script.pyc
monty@python:~/tmp$
```

RUNNABLE SCRIPTS UNDER LINUX

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

So far, we have started our Python scripts with

```
python3 my_file.py
```

on the bash command line.

A Python script can also be started like any other script under Linux, e.g. Bash scripts. Two steps are necessary for this purpose:

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

```
$ chmod +x my_first_simple_script.py
```

We illustrate this in a bash session:

```
bernd@saturn: $ more my_first_simple_script.py
#!/usr/bin/env python3
print("My first simple Python script!")
bernd@saturn: $ ls -l my_first_simple_script.py
-rw-rw-r-- 1 bernd bernd 40 Feb  4 09:32 my_first_simple_script.py
bernd@saturn: $ chmod +x my_first_simple_script.py
bernd@saturn: $ ls -l my_first_simple_script.py
-rwxrwxr-x 1 bernd bernd 40 Feb  4 09:32 my_first_simple_script.py
My first simple Python script!
```

DIFFERENCES BETWEEN COMPILERS AND INTERPRETERS

COMPILER

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

INTERPRETER

Definition: An interpreter is a computer program that executes instructions written in a programming language. It can either

- execute the source code directly or
- translates the source code in a first step into a more efficient representation and executes this code

STRUCTURING WITH INDENTATION

BLOCKS

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



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

The first time, block structures had been formalized was in ALGOL, called a compound statement.

Programming languages usually use certain methods to group statements into blocks:

- begin ... end
ALGOL, Pascal and others
An code snippet in Pascal to show this usage of blocks:

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

- do ... done
e.g. Bourne and Bash shell

- Braces (also called curly brackets): { ... }

By far the most common approach, used by C, C++, Perl, Java, and many other programming languages are braces.

The following examples shows a conditional statement in C:

```
if (x==42) {
    printf("The Answer to the Ultimate Question of Life, the
    Universe, and Everything\n");
} else {
    printf("Just a number!\n");
}
```

The indentations in this code fragment are not necessary. So the code could be written

- offending common decency - as

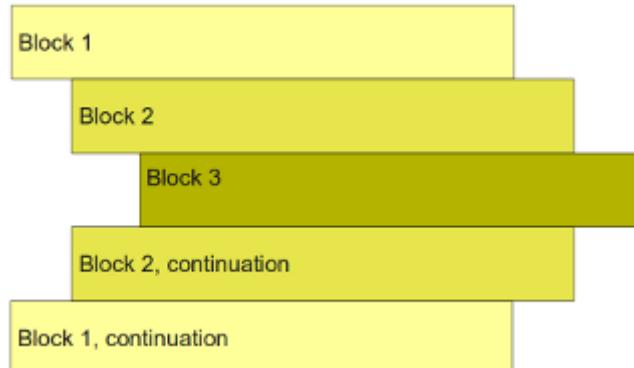
```
if (x==42) {printf("The Answer to the Ultimate Question of
Life, the Universe, and Everything\n");} else {printf("Just a
number!\n");}
```

Please, keep this in mind to understand the advantages of Python!

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

INDENTING CODE

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



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

Beginners are not supposed to understand the following example, because we haven't introduced most of the used structures, like conditional statements and loops. Please confer the following chapters about loops and conditional statements for explanations.

The program implements an algorithm to calculate Pythagorean triples. You will find an explanation of the Pythagorean numbers in our chapter on **for loops**.

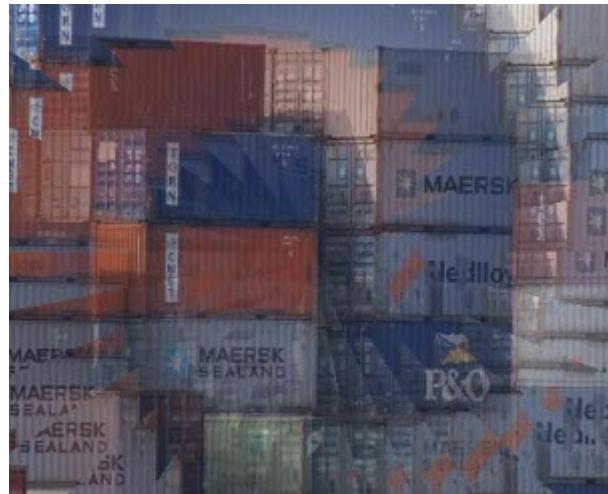
```
from math import sqrt
n = input("Maximum Number? ")
n = int(n)+1
for a in range(1,n):
    for b in range(a,n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print(a, b, c)
```

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

DATA TYPES AND VARIABLES

INTRODUCTION

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



If you want to use lists or associative arrays in C e.g., you will have to construe the data type list or associative arrays from scratch, i.e. design memory structure and the allocation management. You will have to implement the necessary search and access methods as well. Python provides power data types like lists and associative arrays, called dict in Python, as a genuine part of the language.

So, this chapter is worth reading both for beginners and for advanced programmers in other programming languages.

VARIABLES

GENERAL CONCEPT OF VARIABLES

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

So, what's a variable? As the name implies, a variable is something which can change. A variable is a way of referring to a memory location used by a computer program. In many

in programming languages a variable is a symbolic name for this physical location. This memory location contains values, like numbers, text or more complicated types. We can use this variable to tell the computer to save some data in this location or to retrieve some data from this location.

A variable can be seen as a container (or some say a pigeonhole) to store certain values. While the program is running, variables are accessed and sometimes changed, i.e. a new value will be assigned to a variable.

What we have said so far about variables fits best to the way variables are implemented in C, C++ or Java. Variable names have to be declared in these languages before they can be used.

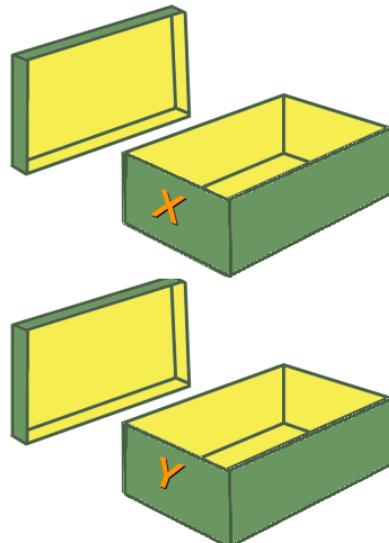
```
int x;  
int y;
```

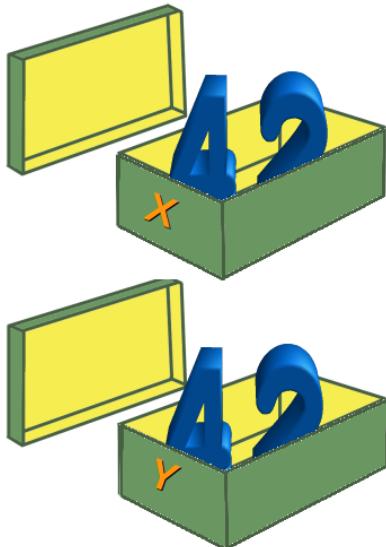
Such declarations make sure that the program reserves memory for two variables with the names x and y. The variable names stand for the memory location. It's like the two empty shoeboxes, which you can see in the picture on the right side. These shoeboxes are labelled with x and y. Like the two shoeboxes,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

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

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

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

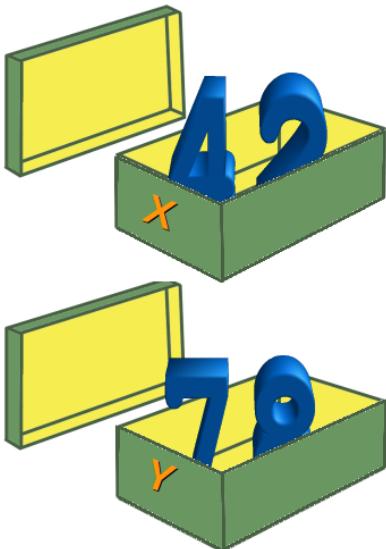




We think that it is easy to understand what happens, if we assign a new value to one of the variables, let's say the values 78 to y.

```
y = 78;
```

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



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

VARIABLES IN PYTHON

It's a lot easier in Python. There is no declaration of variables required in Python. It's not even possible. If there is need of a variable, you think of a name and start using it as a variable.

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

In the following line of code, we assign the value 42 to a variable:

```
i = 42
```

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

```
>>> i = i + 1
>>> print(i)
43
>>>
```

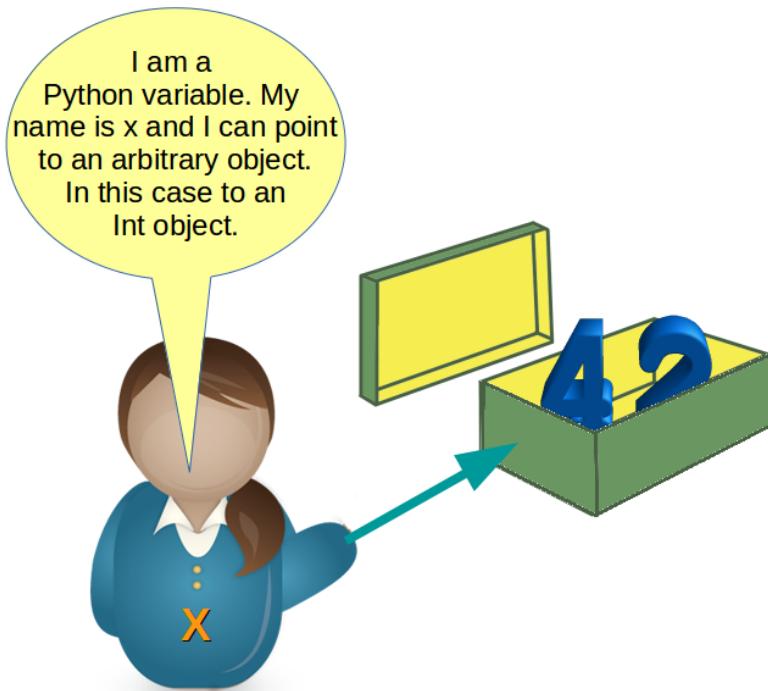
As we have said above, the type of a variable can change during the execution of a script. Or to be precise: A new object, which can be of any type, will be assigned to it. We illustrate this in our following example:

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

Python automatically takes care of the physical representation for the different data types, i.e. an integer values will be stored in a different memory location than a float or a string.

OBJECT REFERENCES

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



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

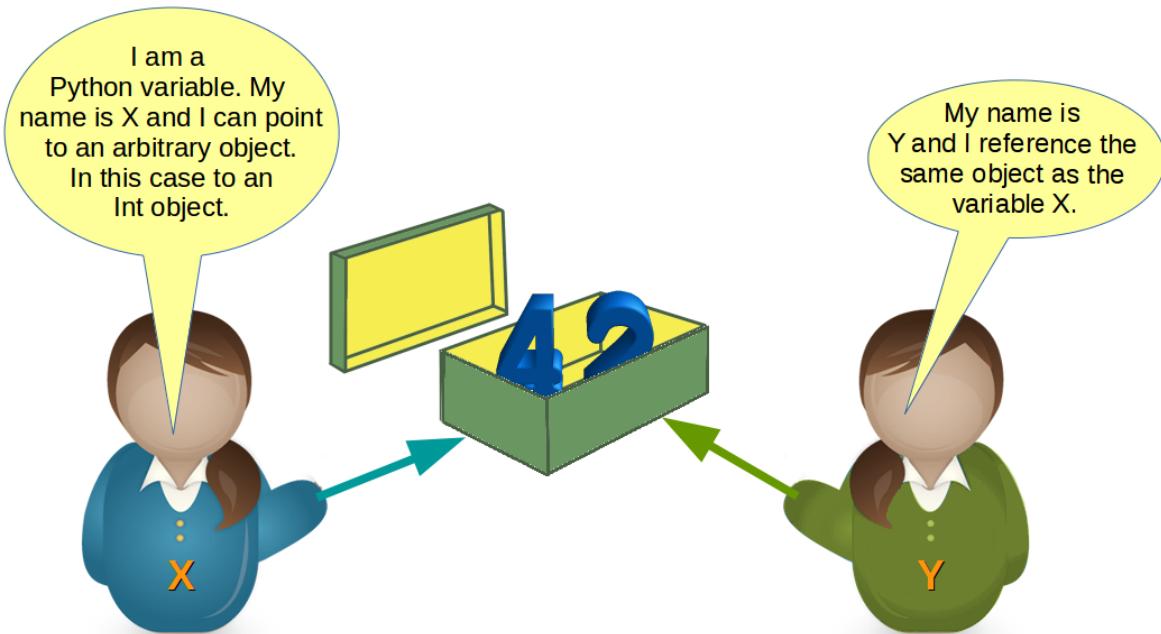
Therefore it is possible to write code like the following in Python:

```
>>> x = 42
>>> print(x)
42
>>> x = "Now x references a string"
>>> print(x)
Now x references a string
```

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

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

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

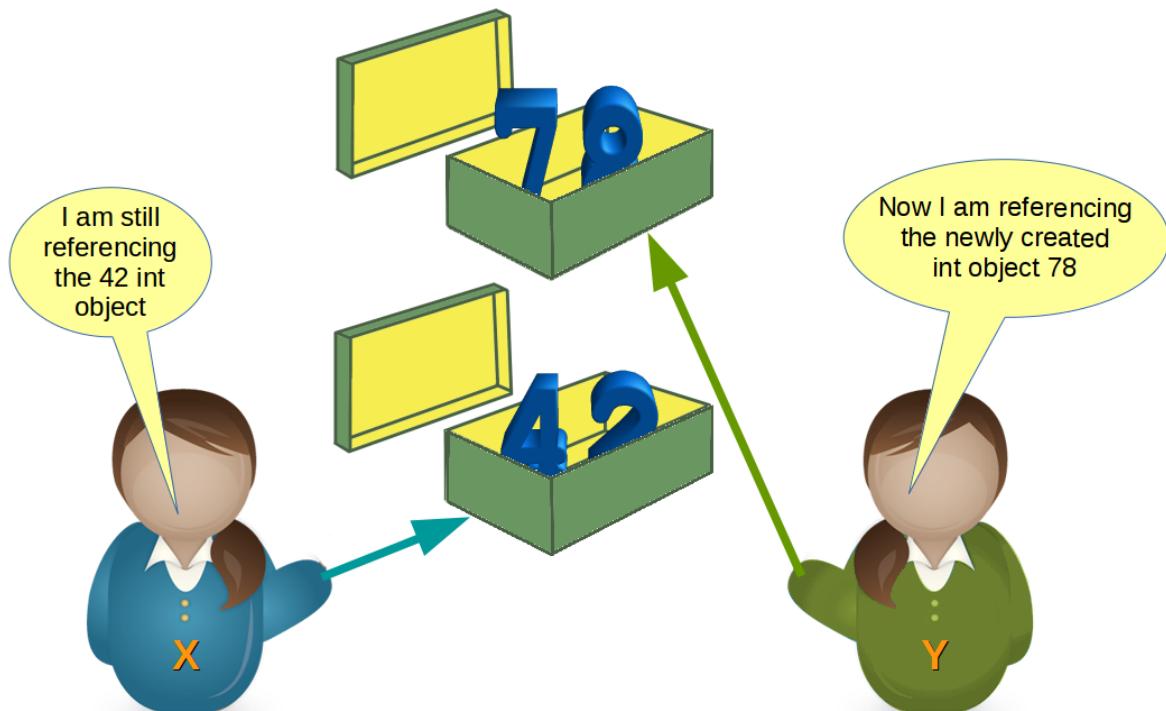


What will happen, when we execute

```
y = 78
```

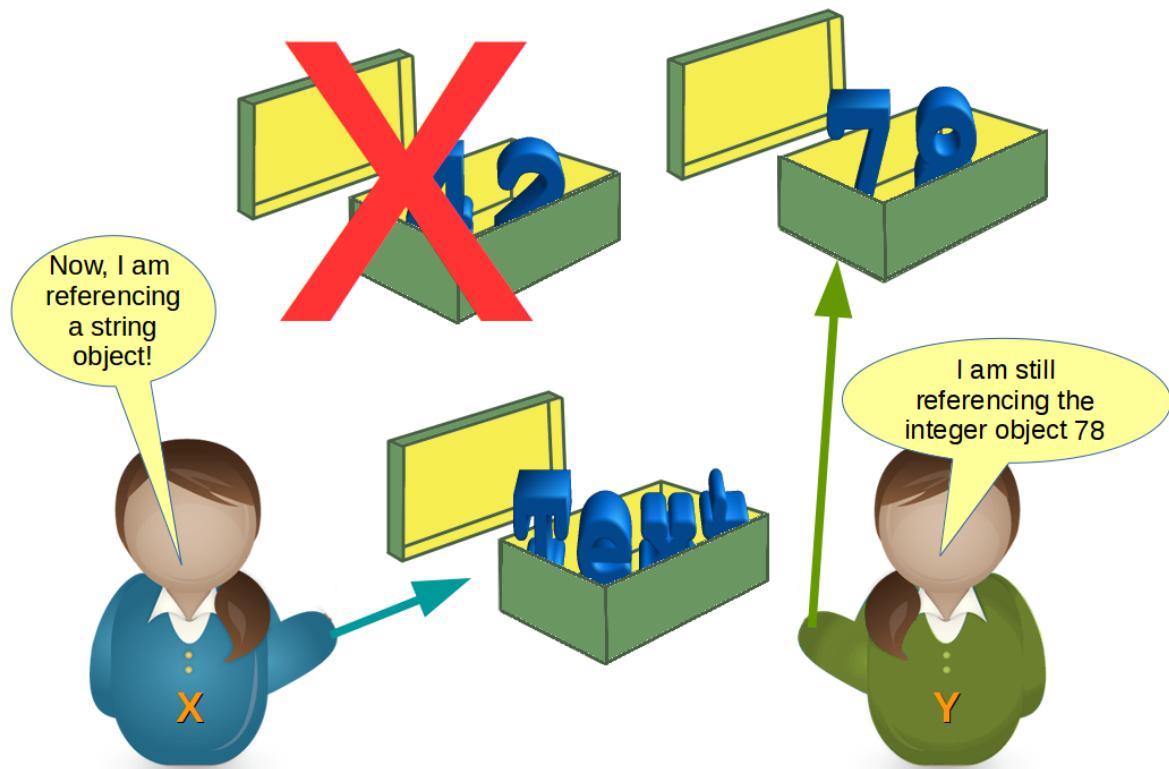
after the previous code?

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



Most probably, we will see further changes to the variables in the flow of our program.

There might be, for example, a string assignment to the variable `x`. The previously integer object "42" will be orphaned after this assignment. It will be removed by Python, because no other variable is referencing it.



ID FUNCTION

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

The identity function `id()` can be used for this purpose. Every instance (object or variable) has an identity, i.e. an integer which is unique within the script or program, i.e. other objects have different identities.

So, let's have a look at our previous example and how the identities will change:

```
>>> x = 42
>>> id(x)
10107136
>>> y = x
>>> id(x), id(y)
(10107136, 10107136)
>>> y = 78
>>> id(x), id(y)
(10107136, 10108288)
>>>
```

VALID VARIABLE NAMES

The naming of variables follows the more general concept of an identifier. A Python identifier is a name used to identify a variable, function, class, module or other object. A variable name and an identifier can consist of the uppercase letters "A" through "Z", the lowercase letters "a" through "z", the underscore _ and, except for the first character, the digits 0 through 9. Python 3.x is based on Unicode. This means that variable names and identifier names can additionally contain Unicode characters as well. Identifiers are unlimited in length. Case is significant. The fact that identifier names are case-sensitive can cause problems to some Windows users, where file names are case-insensitive, for example.

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

The following variable definitions are all valid:

```
>>> height = 10
>>> maximum_height = 100
>>>
>>> υψος = 10
>>> μεγιστη_υψος = 100
>>>
>>> MinimumHeight = 1
```

PYTHON KEYWORDS

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

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

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

```
>>> help()
```

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://docs.python.org/3.4/tutorial/>.

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

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

help>

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

help> keywords

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

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

help>

NAMING CONVENTIONS

We saw in our chapter on "Valid Variable Names" that we sometimes need names which consist of more than one word. We used, for example, the name "maximum_height". The

underscore functioned as a word separator, because blanks are not allowed in variable names. Some people prefer to write Variable names in the so-called CamelCase notation. We defined the variable MinimumHeight in this naming style.

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

Certain names should be avoided for variable names: Never use the characters 'l' (lowercase letter "L"), 'O' ("O" like in "Ontario"), or 'T' (like in "Indiana") as single character variable names. They should be avoided, because these characters are indistinguishable from the numerals one and zero in some fonts. When tempted to use 'l', use 'L' instead, if you cannot think of a better name anyway. The Style Guide has to say the following about the naming of identifiers in standard modules:

"All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names."

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

CHANGING DATA TYPES AND STORAGE LOCATIONS

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

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

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

NUMBERS

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

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

```
>>> a = 0o10
>>> print(a)
8
```
 - Hexadecimal literals (base 16)
Hexadecimal literals have to be prefixed either by "0x" or "0X".
example:

```
>>> hex_number = 0xA0F
>>> print(hex_number)
2575
```
 - Binary literals (base 2)
Binary literals can easily be written as well. They have to be prefixed by a leading "0", followed by a "b" or "B":

```
>>> x = 0b101010 >>> x 42 >>>
```

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

```
>>> x = hex(19)
>>> x
'0x13'
>>> type(x)
<class 'str'>
>>> x = bin(65)
>>> x
'0b1000001'
>>> x = oct(65)
>>> x
'0o101'
>>> oct(0b101101)
'0o55'
>>>
```

Integers in Python3 can be of unlimited size

```
>>> x = 787366098712738903245678234782358292837498729182728
>>> print(x)
787366098712738903245678234782358292837498729182728
>>> x * x * x
4881239700706382159867701621057313155388275860919486179978711229
5022889112396090191830861828631152328223931370827558978712300531
```

```
7148968569797875581092352
>>>
```

- Long integers

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

```
>>> 1L
  File "<stdin>", line 1
    1L
    ^
SyntaxError: invalid syntax
>>> x = 43
>>> long(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'long' is not defined
>>>
```

- Floating-point numbers

for example: 42.11, 3.1415e-10

- Complex numbers

Complex numbers are written as *<real part> + <imaginary part>j*

examples:

```
>>> x = 3 + 4j
>>> y = 2 - 3j
>>> z = x + y
>>> print(z)
(5+1j)
```

INTEGER DIVISION

There are two kinds of division operators:

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

TRUE DIVISION

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

```
bernd@marvin ~/dropbox/websites/python-course.eu $ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 10 / 3
3.333333333333335
>>> 10.0 / 3.0
3.333333333333335
>>> 10.5 / 3.5
3.0
>>>
```

FLOOR DIVISION

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

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

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

```
bernd@marvin $ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 9 // 3
3
>>> 10 // 3
3
>>> 11 // 3
3
>>> 12 // 3
4
>>> 10.0 // 3
3.0
>>> -7 // 3
-3
>>> -7.0 // 3
```

-3.0

>>>

STRINGS

The task of the first-generation computers in the forties and fifties had been - due to technical restraints - focussed on number processing. Text processing had been just a dream that time. Nowadays, one of the main tasks of computers is text processing in all its varieties; the most prominent applications are search engines like Google. To enable text processing programming

languages need suitable data types. Strings are used in all modern programming languages to store and process textual information. Logically, a string - like any text - is a sequence of characters. The question remains what a character consists of. In a book, or in a text like the one you are reading now, characters consist of graphical shapes, so-called graphemes, consisting of lines, curves and crossings in certain angles or positions and so on. The ancient Greeks associated with the word the engraving on coins or the stamps on seals.

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

ASCII is restricted to 128 characters and "Extended ASCII" is still limited to 256 bytes or characters. This is good enough for languages like English, German and French, but by far not sufficient for Chinese, Japanese and Korean. That's where Unicode gets into the



game. Unicode is a standard designed to represent every character from every language, i.e. it can handle any text of the world's writing systems. These writing systems can also be used simultaneously, i.e. Roman alphabet mixed with Cyrillic or even Chinese characters.

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

You can transform it like this in Python by the way:

```
>>> hex(65)
'0x41'
>>> int(0x41)
65
```

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

UNICODE ENCODINGS	
Name	Description
UTF-32	It's a one to one encoding, i.e. it takes each Unicode character (a 4-byte number) and stores it in 4 bytes. One advantage of this encoding is that you can find the Nth character of a string in linear time, because the Nth character starts at the $4 \times N$ th byte. A serious disadvantage of this approach is due to the fact that it needs four bytes for every character.
UTF-16	Hardly anybody needs more than 65535 characters, so UTF-16, which needs 2 bytes, is a more space efficient alternative to UTF-32. But it is very difficult to access characters outside the range 0 - 65535, i.e. characters from the so-called "astral plane". Another problem of both UTF-32 and UTF-16 consists in the byte ordering, which is depending on the operating system.
UTF-8	UTF8 is a variable-length encoding system for Unicode, i.e. different characters take up a different number of bytes. ASCII characters use solely one byte per character, which are even the same as the used to be for the first 128 characters (0 - 127), i.e. these characters in UTF-8 are indistinguishable from ASCII. But the so-called "Extended Latin" characters like the Umlaute ä, ö and so on take up two bytes.

bytes. Chinese characters need three bytes. Finally, the very seldom used characters of the "astral plane" need four bytes to be encoded in UTF-8. A Disadvantage of this approach is that finding the Nth character is more complex, the longer the string, the longer it takes to find a specific character.

STRING, UNICODE AND PYTHON

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

There are different ways to define strings in Python:

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

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

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

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

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

Analogously, we will have to escape a double quote inside a double quoted string:

```
>>> txt = "He said: \"It doesn't matter, if you enclose a string in  
single or double quotes!\""  
>>> print(txt)  
He said: "It doesn't matter, if you enclose a string in single or  
double quotes!"  
>>>
```

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

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

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A "quote" is the character

used to open the string, i.e. either ' or ".)

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

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

```
>>> s = "Hello World"
>>> s[0]
'H'
>>> s[5]
' '
>>>
```

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

```
>>> s[len(s)-1]
'd'
```

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

```
>>> s[-1]
'd'
>>> s[-2]
'l'
>>>
```

Some readers might find it confusing, when we use "to subscript" as a synonym for "to index". Usually, a subscript is a number, figure, symbol, or indicator that is smaller than the normal line of type and is set slightly below or above it. When we wrote s_0 or s_3 in the previous examples, this can be seen as an alternative way for the notation s_0 or s_3 . So, both s_3 and $s[3]$ describe or denote the 4th character. Btw. there exists no character type in Python. A character is simply a string of size one.

P	y	t	h	o	n
0	1	2	3	4	5

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



SOME OPERATORS AND FUNCTIONS FOR STRINGS

- **Concatenation**

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

- **Repetition**

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

- **Indexing**

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

- **Slicing**

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



- **Size**

len("Python") will result in 6

IMMUTABLE STRINGS

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

```
>>> s = "Some things are immutable!"
>>> s[-1] = "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

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

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

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

A STRING PECULIARITY

Strings show a special effect, which we will illustrate in the following example. We will need the "is"-Operator. If both a and b are strings, "a is b" checks if they have the same identity, i.e. share the same memory location. If "a is b" is True, then it trivially follows that "a == b" has to be True as well.

Yet, "a == b" True doesn't imply that "a is b" is True as well!

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

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

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

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

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

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

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

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

ESCAPE SEQUENCES IN STRINGS

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

Escape Sequence	Meaning Notes
\newline	Ignored
\\\	Backslash (\)
\'	Single quote ('')
\"	Double quote ("")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxxx	Character with 32-bit hex valuexxxxxxxx (Unicode only)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

BYTE STRINGS

Python 3.0 uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. Every string or text in Python 3 is Unicode, but encoded Unicode is represented as binary data. The type used to hold text is str, the type used to hold data is bytes. It's not possible to mix text and data in Python 3; it will raise TypeError.

While a string object holds a sequence of characters (in Unicode), a bytes object holds a sequence of bytes, out of the range 0 .. 255, representing the ASCII values.

Defining bytes objects and casting them into strings:

```
>>> x = b"Hello"  
>>> t = str(x)  
>>> u = t.encode("UTF-8")
```

ARITHMETIC AND COMPARISON OPERATORS

INTRODUCTION

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

OPERATORS

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

Operator	Description	Example
+, -	Addition, Subtraction	10 -3
*, %	Multiplication, Modulo	27 % 7 Result: 6
/	Division This operation results in different results for Python 2.x (like floor division) and Python 3.x	Python3: <pre>>>> 10 / 3 3.3333333333333335</pre> and in Python 2.x: <pre>>>> 10 / 3 3</pre>
//	Truncation Division (also known as floordivision or floor division) The result of this division is the integral part of the result, i.e. the fractional part is truncated, if there is any. It works both for integers and floating-point numbers, but there is a difference in the type of the results: If both the dividend and the divisor are integers, the result will be also an integer. If either the dividend or the divisor is a float, the result will be the truncated result as a float.	 <pre>>>> 10 // 3 3</pre> If at least one of the operands is a float value, we get a truncated float value as the result. <pre>>>> 10.0 // 3 3.0 >>></pre> A note about efficiency: The results of <code>int(10 / 3)</code> and <code>10 // 3</code> are equal. But the "://" division is more than two times as fast! You can see this here: <pre>In [9]: %%timeit for x in range(1,</pre>

		<pre> 100): y = int(100 / x) ... 100000 loops, best of 3: 11.1 µs per loop In [10]: %timeit for x in range(1, 100): y = 100 // x ... 100000 loops, best of 3: 4.48 µs per loop </pre>
+x, -x	Unary minus and Unary plus (Algebraic signs)	-3
~x	Bitwise negation	~3 - 4 Result: -8
**	Exponentiation	10 ** 3 Result: 1000
or, and, not	Boolean Or, Boolean And, Boolean Not	(a or b) and c
in	"Element of"	1 in [3, 2, 1]
<, <=, >, >=, !=, ==	The usual comparison operators	2 <= 3
, &, ^	Bitwise Or, Bitwise And, Bitwise XOR	6 ^ 3
<<, >>	Shift Operators	6 << 3

SEQUENTIAL DATA TYPES

INTRODUCTION

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

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



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

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

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

Accessing lists:

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

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

```
>>> countries =
["Germany", "Switzerland", "Austria", "France", "Belgium",
```

```
"Netherlands", "England"]  
>>> len(countries)  
7  
>>> fib = [1,1,2,3,5,8,13,21,34,55]  
>>> len(fib)  
10  
>>>
```

BYTES

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

PYTHON LISTS

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

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

The main properties of Python lists:

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

LIST NOTATION AND EXAMPLES

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



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

ACCESSING LIST ELEMENTS

Assigning a list to a variable:

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

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

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

The previous example of a list has been a list with elements of equal data types. But as we had before, lists can have various data types. The next example shows this:

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

SUBLISTS

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

```
>>> person = [["Marc", "Mayer"], ["17, Oxford Str",
"12345", "London"], "07876-7876"]
>>> name = person[0]
>>> print(name)
['Marc', 'Mayer']
>>> first_name = person[0][0]
>>> print(first_name)
Marc
>>> last_name = person[0][1]
>>> print(last_name)
Mayer
>>> address = person[1]
>>> street = person[1][0]
>>> print(street)
17, Oxford Str
```

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

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

TUPLES

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

Where is the benefit of tuples?

- Tuples are faster than lists.
- If you know that some data doesn't have to be changed, you should use tuples instead of lists, because this protects your data against accidental changes.
- The main advantage of tuples consists in the fact that tuples can be used as keys in dictionaries, while lists can't.

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

```
>>> t = ("tuples", "are", "immutable")
>>> t[0]
'tuples'
>>> t[0] = "assignments to elements are not possible"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

SLICING

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

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

```
>>> str = "Python is great"
>>> first_six = str[0:6]
>>> first_six
'Python'
>>> starting_at_five = str[5:]
>>> starting_at_five
'n is great'
>>> a_copy = str[:]
>>> without_last_five = str[0:-5]
>>> without_last_five
'Python is '
```

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

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

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

```
>>> all_but_last_two = cities[:-2]
>>> print(all_but_last_two)
['Vienna', 'London', 'Paris', 'Berlin']
>>>
```

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

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

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

The resulting sequence consists of the following elements:

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

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

```
>>> str = "Python under Linux is great"
>>> str[::3]
'Ph d n e'
```

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

```
"TPoyrtohnotno ciosu rtshees lianr gTeosrto nCtiot yb yi nB
oCdaennasdeao"
```

Try to figure it out using slicing with the step argument. The solution is: You have to set step to 2

```
>>> s
'TPoyrtohnotno ciosu rtshees lianr gTeosrto nCtiot yb yi nB
oCdaennasdeao'
>>> s[::2]
'Toronto is the largest City in Canada'
>>> s[1::2]
'Python courses in Toronto by Bodenseo'
>>>
```

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

```
>>> s = "Toronto is the largest City in Canada"
>>> t = "Python courses in Toronto by Bodenseo"
>>> s = "".join(["".join(x) for x in zip(s,t)])
>>> s
'TPoyrtohnotno ciosu rtshees lianr gTeosrto nCtiot yb yi nB
oCdaennasdeao'
>>>
```

LENGTH



The length of a sequence, i.e. a list, a string or a tuple, can be determined with the function `len()`. For strings it counts the number of characters and for lists or tuples the number of elements are counted, whereas a sublist counts as 1 element.

```
>>> txt = "Hello World"
>>> len(txt)
11
>>> a = ["Swen", 45, 3.54, "Basel"]
>>> len(a)
```

4

CONCATENATION OF SEQUENCES

Combining two sequences like strings or lists is as easy as adding two numbers. Even the operator sign is the same.

The following example shows how to concatenate two strings into one:

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

It's as simple for lists:

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

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

```
s += t
```

is syntactically the same as:

```
s = s + t
```

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

CHECKING IF AN ELEMENT IS CONTAINED IN LIST

It's easy to check, if an item is contained in a sequence. We can use the "in" or the "not in" operator for this purpose.

The following example shows how this operator can be applied:

```
>>> 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 easy!"
>>> "y" in str
True
>>> "x" in str
False
>>>
```

REPETITIONS

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

It's a kind of abbreviation for an n -times concatenation, i.e.

```
str * 4
```

is the same as

```
str + str + str + str
```

Further examples:

```
>>> 3 * "xyz-"
'xyz-xyz-xyz-'
>>> "xyz-" * 3
```

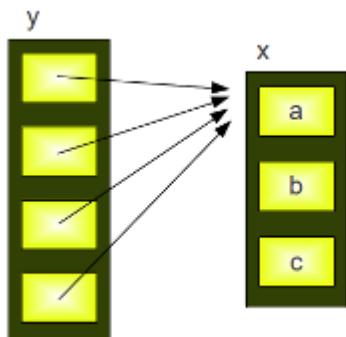
```
'xyz-xyz-xyz-'
>>> 3 * ["a", "b", "c"]
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

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

THE PITFALLS OF REPETITIONS

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

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



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

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

LISTS

CHANGING LISTS

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

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



- **push**

This method is used to put a new object on the stack. Depending on the point of view, we say that we "push" the object on top or attach it to the right side. Python doesn't offer - contrary to other programming languages - no method with the name "push", but the method "append" has the same functionality.

- **pop**

This method returns the most upper element of the stack. The object will be removed from the stack as well.

- **peek**

Some programming languages provide another method, which can be used to view what is on the top of the stack without removing this element. The Python list class doesn't possess such a method, because it is not needed. A peek can be simulated by accessing the element with the index -1:

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

POP AND APPEND

- `s.append(x)`

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

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

It's import to understand that append returns "None". This means that it usually doesn't make sense to reassign the return value:

```
>>> lst = [3, 5, 7]
>>> lst = lst.append(42)
>>> print(lst)
None
>>>
```

- `s.pop(i)`

'pop' returns the *i*th element of a list *s*. The element will be removed from the list as well.

```
>>> cities = ["Hamburg", "Linz", "Salzburg", "Vienna"]
>>> cities.pop(0)
'Hamburg'
>>> cities
['Linz', 'Salzburg', 'Vienna']
>>> cities.pop(1)
'Salzburg'
>>> cities
['Linz', 'Vienna']
>>>
```

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

- `s.pop()`

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

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

EXTEND

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

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

What we wanted to accomplish is the following:

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

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

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

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

```
>>> lst = ["a", "b", "c"]
>>> programming_language = "Python"
>>> lst.extend(programming_language)
>>> print(lst)
['a', 'b', 'c', 'P', 'y', 't', 'h', 'o', 'n']
```

Now with a tuple:

```
>>> lst = ["Java", "C", "PHP"]
>>> t = ("C#", "Jython", "Python", "IronPython")
>>> lst.extend(t)
>>> lst
['Java', 'C', 'PHP', 'C#', 'Jython', 'Python', 'IronPython']
>>>
```

EXTENDING AND APPENDING LISTS WITH THE '+'-OPERATOR

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

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

Be careful. Never ever do the following:

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

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

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

The augmented assignment (+=) is an alternative:

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

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

```
import time

n= 100000

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

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 returns shocking results:

```

26.3175041676
0.0305399894714
0.0207479000092

```

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

REMOVING AN ELEMENT WITH REMOVE

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

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

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

```

>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.remove("green")
>>> colours
['red', 'blue', 'green', 'yellow']
>>> colours.remove("green")
>>> colours
['red', 'blue', 'yellow']
>>> colours.remove("green")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>

```

FIND THE POSITION OF AN ELEMENT IN A LIST

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

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

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

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.index("green")
1
>>> colours.index("green", 2)
3
>>> colours.index("green", 3, 4)
3
>>> colours.index("black")
Traceback (most recent call last):
  File "", line 1, in
ValueError: 'black' is not in list
>>>
```

INSERT

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

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

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

```
>>> lst = ["German is spoken", "in Germany,", "Austria",
          "Switzerland"]
>>> lst.insert(3, "and")
>>> lst
['German is spoken', 'in Germany,', 'Austria', 'and', 'Switzerland']
>>>
```

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

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

Footnotes:

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

SHALLOW AND DEEP COPY

INTRODUCTION

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



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

```
>>> x = 3
>>> y = x
>>> print(id(x), id(y))
9251744 9251744
>>> y = 4
>>> print(id(x), id(y))
9251744 9251776
>>> print(x,y)
3 4
>>>
```

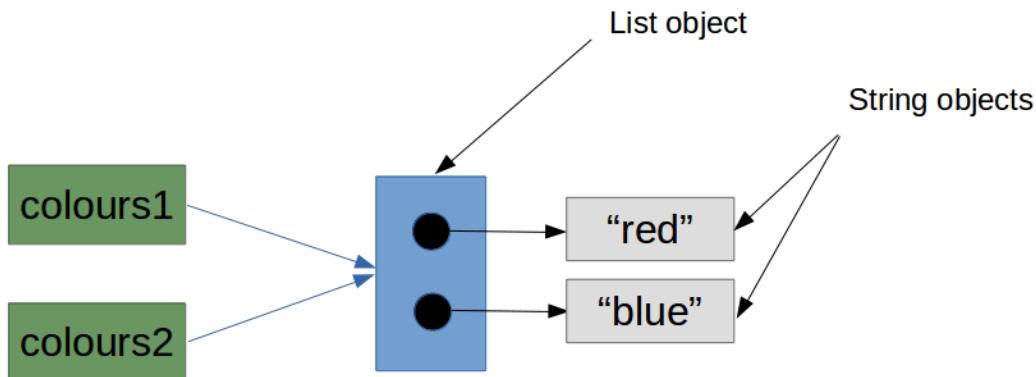
But even if this internal behaviour appears strange compared to programming languages like C, C++ and Perl, yet the observable results of the assignments answer our expectations. But it can be problematic, if we copy mutable objects like lists and dictionaries.

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

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

COPYING A LIST

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1
>>> print(colours1)
['red', 'blue']
>>> print(colours2)
['red', 'blue']
>>> print(id(colours1), id(colours2))
43444416 43444416
>>> colours2 = ["rouge", "vert"]
>>> print(colours1)
['red', 'blue']
>>> print(colours2)
['rouge', 'vert']
>>> print(id(colours1), id(colours2))
43444416 434444200
>>>
```



In the example above a simple list is assigned to colours1. This list is a so-called "shallow list", because it doesn't have a nested structure, i.e. no sublists are contained in the list. In the next step we assign colour1 to colours2.

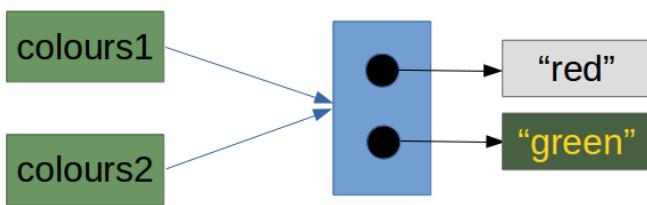
The `id()` function shows us that both variables point to the same list object, i.e. they share this object.

Now we want to see, what happens, if we assign a new list object to colours2.

As we have expected, the values of colours1 remained unchanged. Like it was in our example in the chapter "Data types and variables" a new memory location had been allocated for colours2, because we have assigned a completely new list, i.e. a new list object to this variable. The picture on the left side needs some explanation as well: We have two variable names "colours1" and "colours2", which we have depicted as green boxes. The blue box symbolizes the list object. A list object consists of references to other objects. In our example the list object, which is references by both variables, references two string objects, i.e. "red" and "blue".

Now we have to examine, what will happen, if we change just one element of the list of colours2 or colours1:

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1
>>> print(id(colours1), id(colours2))
14603760 14603760
>>> colours2[1] = "green"
>>> print(id(colours1), id(colours2))
14603760 14603760
>>> print(colours1)
['red', 'green']
>>> print(colours2)
['red', 'green']
>>>
```



Let's see, what has happened in detail in the previous code. We assigned a new value to the second element of colours2, i.e. the element with the index 1. Lots of beginners will be stunned that the list of colours1 has been "automatically"

changed as well. Of course, we don't have two lists: We have only two names for the same list!

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

COPY WITH THE SLICE OPERATOR

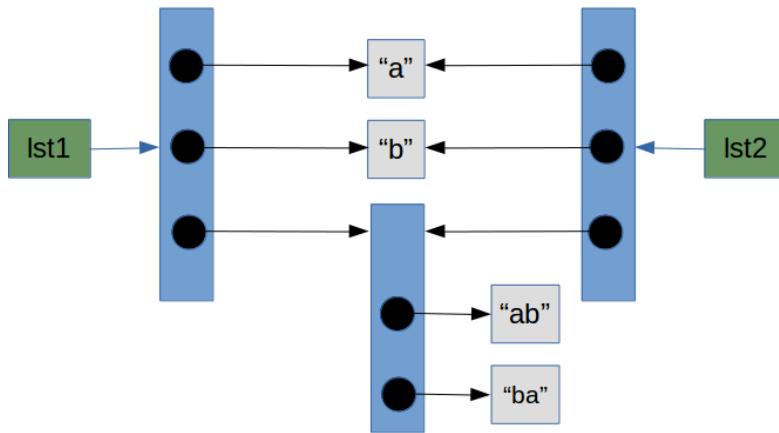
It's possible to completely copy shallow list structures with the slice operator without having any of the side effects, which we have described above:

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

But as soon as a list contains sublists, we have another difficulty: The sublists are not copied but only the references to the sublists. The following example list "lst2" contains one sublist. We create a shallow copy with the slicing operator.

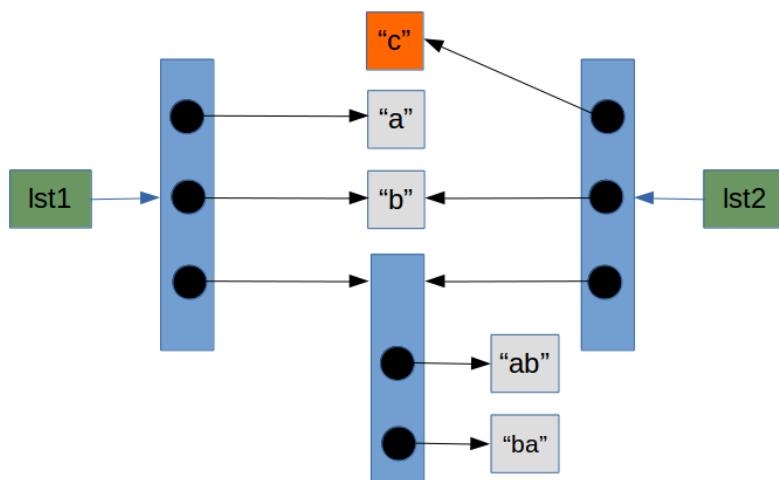
```
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>> lst2 = lst1[:]
```

The following diagram depicts the data structure after the copying. We can see that both `lst1[2]` and `lst2[2]` point to the same object, i.e. the sublist:



If you assign a new value to the 0th or the 1st index of one of the two lists, there will be no side effect.

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

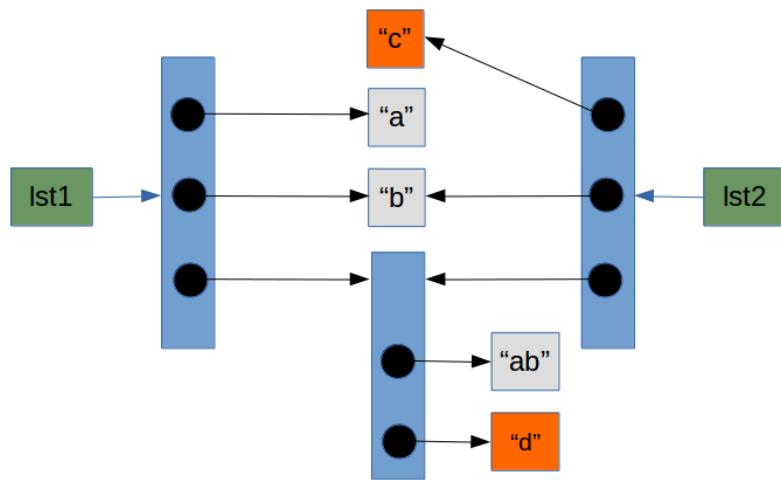


Problems arise, if you change one

of the elements of the sublist:

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

The following diagram depicts the situation after we have executed the code above. We can see that both lst1 and lst2 are affected by the assignment lst2[2][1] = 'd':



USING THE METHOD DEEPCOPY FROM THE MODULE COPY

A solution to the described problems provide the module "copy". This module provides the method "deepcopy", which allows a complete or deep copy of an arbitrary list, i.e. shallow and other lists.

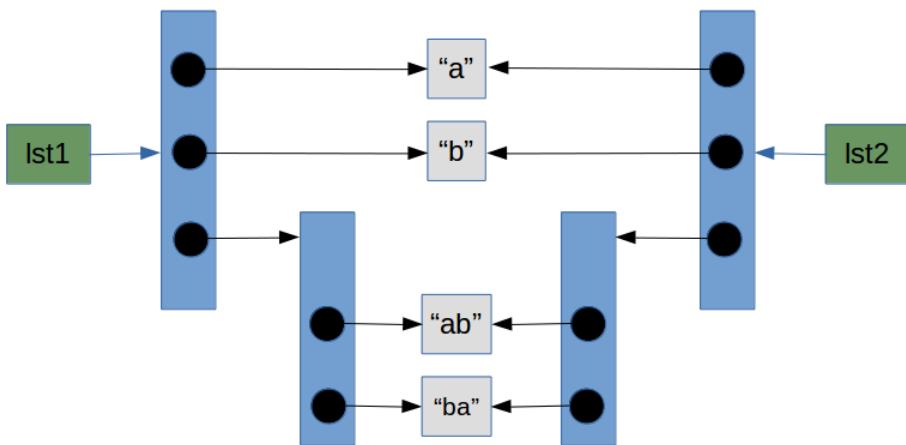
Let's use deepcopy for our previous list:

```

>>> from copy import deepcopy
>>>
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>>
>>> lst2 = deepcopy(lst1)
>>>
>>> lst1
['a', 'b', ['ab', 'ba']]
>>> lst2
['a', 'b', ['ab', 'ba']]
>>> id(lst1)
139716507600200
>>> id(lst2)
139716507600904
>>> id(lst1[0])
139716538182096
>>> id(lst2[0])
139716538182096
>>> id(lst2[2])
139716507602632
>>> id(lst1[2])
  
```

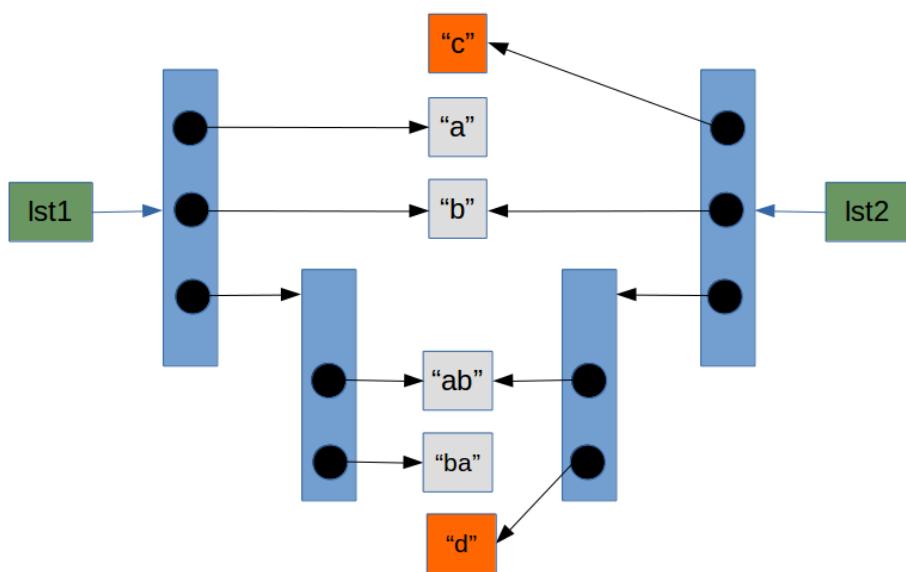
```
139716507615880
>>>
```

We can see by using the id function that the sublist has been copied, because `id(lst2[2])` is different from `id(lst1[2])`. An interesting fact is that the strings are not copied: `lst1[0]` and `lst2[0]` reference the same string. This is true for `lst1[1]` and `lst2[1]` as well, of course. The following diagram shows the situation after copying the list:



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

Now the data structure looks like this:



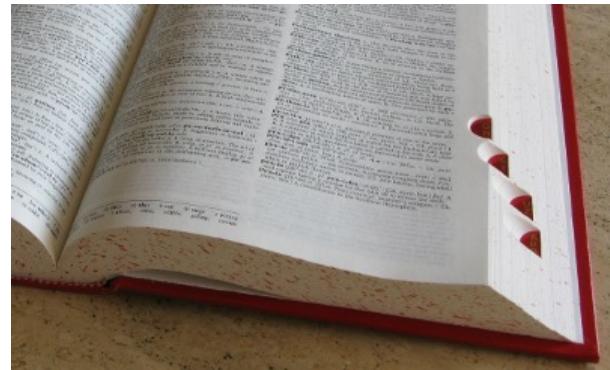
© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

DICTIONARIES

INTRODUCTION

We have already become acquainted with lists in the previous chapter. In this chapter of our online Python course we will present the dictionaries and the operators and methods on dictionaries. Python programs or scripts without lists and dictionaries are nearly inconceivable. Dictionaries and their powerful implementations are part of what makes Python so effective and superior.

Like lists they can be easily changed, can be shrunk and grown ad libitum at run time. They shrink and grow without the necessity of making copies. Dictionaries can be contained in lists and vice versa.



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

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

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

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

EXAMPLES OF DICTIONARIES

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

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

If we want to get the population of one of those cities, all we have to do is to use the name of the city as an index:

```
>>> city_population["New York City"]
8550405
>>> city_population["Toronto"]
2731571
>>> city_population["Boston"]
667137
```

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

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

If you look at the way, we have defined our dictionary, you might get the impression that we have an ordering in our dictionary, i.e. the first one "New York City", the second one "Los Angeles" and so on. But be aware of the fact that there is no ordering in dictionaries. That's the reason, why the output of the city dictionary, doesn't reflect the "original ordering":

```
>>> city_population
{'Toronto': 2615060, 'Ottawa': 883391, 'Los Angeles': 3792621,
'Chicago': 2695598, 'New York City': 8175133, 'Boston': 62600,
'Washington': 632323, 'Montreal': 11854442}
```

Therefore it is neither possible to access an element of the dictionary by a number, like we did with lists:

```
>>> city_population[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 0
>>>
```

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

```
>>> city_population["Halifax"] = 390096
>>> city_population
{'Toronto': 2615060, 'Ottawa': 883391, 'Los Angeles': 3792621,
```

```
'Chicago': 2695598, 'New York City': 8175133, 'Halifax': 390096,
'Boston': 62600, 'Washington': 632323, 'Montreal': 11854442}
>>>
```

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

```
>>> city = {}
>>> city
{ }
```

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

```
>>> food = {"ham" : "yes", "egg" : "yes", "spam" : "no" }
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'no'}
>>> food["spam"] = "yes"
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'yes'}
>>>
```

Our next example is a simple English-German dictionary:

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

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

```
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu",
"gelb": "jaune"}
```

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

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

The output of the previous script:

```
{'blue': 'blau', 'green': 'grün', 'yellow': 'gelb', 'red': 'rot'}
rot
```

The French word for red is: rouge

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

```
>>> dic = { [1,2,3] :"abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

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

```
>>> dic = { (1,2,3) :"abc", 3.1415 :"abc"}
>>> dic
{3.1415: 'abc', (1, 2, 3): 'abc'}
```

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

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau",
"yellow": "gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu",
"gelb": "jaune"}

dictionaries = {"en_de" : en_de, "de_fr" : de_fr }
print(dictionaries["de_fr"]["blau"])
```

OPERATORS ON DICTIONARIES

Operator Explanation

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

Examples:

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

```
morse = {
"A" : ".-",
"B" : "-...",
"C" : "-.-.",
"D" : "-..",
"E" : ".,"}
```

```

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

```

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

```
from morsecode import Morse
```

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

```
>>> len(morse)
38
```

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

```
>>> "a" in morse
False
>>> "A" in morse
True
>>> "a" not in morse
True
```

POP() AND POPITEM()

POP

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

```
>>> en_de = {"Austria": "Vienna", "Switzerland": "Bern",
    "Germany": "Berlin", "Netherlands": "Amsterdam"}
>>> capitals = {"Austria": "Vienna", "Germany": "Berlin",
    "Netherlands": "Amsterdam"}
>>> capital = capitals.pop("Austria")
>>> print(capital)
Vienna
>>> print(capitals)
{'Netherlands': 'Amsterdam', 'Germany': 'Berlin'}
>>> capital = capitals.pop("Switzerland")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Switzerland'
>>>
```

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

```
>>> capital = capitals.pop("Switzerland", "Bern")
>>> print(capital)
Bern
>>> capital = capitals.pop("France", "Paris")
>>> print(capital)
Paris
>>> capital = capitals.pop("Germany", "München")
>>> print(capital)
Berlin
>>>
```

POPITEM()

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

KeyError will be raised.

```
>>> capitals = {"Springfield": "Illinois", "Augusta": "Maine",
   "Boston": "Massachusetts", "Lansing": "Michigan", "Albany": "New
   York", "Olympia": "Washington", "Toronto": "Ontario"}
>>> (city, state) = capitals.popitem()
>>> (city, state)
('Olympia', 'Washington')
>>> print(capitals.popitem())
('Albany', 'New York')
>>> print(capitals.popitem())
('Boston', 'Massachusetts')
>>> print(capitals.popitem())
('Lansing', 'Michigan')
>>> print(capitals.popitem())
('Toronto', 'Ontario')
>>>
```

ACCESSING NON EXISTING KEYS

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

```
>>> locations = {"Toronto" : "Ontario", "Vancouver": "British
   Columbia"}
>>> locations["Ottawa"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Ottawa'
```

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

```
>>> if "Ottawa" in locations: print(locations["Ottawa"])
...
>>> if "Toronto" in locations: print(locations["Toronto"])
...
Ontario
```

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

```
>>> proj_language = {"proj1": "Python", "proj2": "Perl",
   "proj3": "Java"}
>>> proj_language["proj1"]
'Python'
>>> proj_language["proj4"]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 'proj4'
>>> proj_language.get("proj2")
```

```
'Perl'
>>> proj_language.get("proj4")
>>> print(proj_language.get("proj4"))
None
>>> # setting a default value:
>>> proj_language.get("proj4", "Python")
'Python'
>>>
```

IMPORTANT METHODS

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

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

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

```
# -*- coding: utf-8 -*-

trainings = { "course1": {"title": "Python Training Course for
Beginners",
                           "location": "Frankfurt",
                           "trainer": "Steve G. Snake"},

              "course2": {"title": "Intermediate Python Training",
                           "location": "Berlin",
                           "trainer": "Ella M. Charming"},

              "course3": {"title": "Python Text Processing Course",
                           "location": "München",
                           "trainer": "Monica A. Snowdon"}}

trainings2 = trainings.copy()

trainings["course2"]["title"] = "Perl Training Course for Beginners"
print(trainings2)
```

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

```
{'course2': {'trainer': 'Ella M. Charming', 'name': 'Perl Training
Course for Beginners', 'location': 'Berlin'}, 'course3': {'trainer':
'Monica A. Snowdon', 'name': 'Python Text Processing Course',
'location': 'München'}, 'course1': {'trainer': 'Steve G. Snake',
'name': 'Python Training Course for Beginners', 'location':
'Frankfurt'}}}
```

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

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

trainings2 = trainings.copy()

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

The statements `print(trainings2["course2"])` outputs still the original Python course:

```
{'trainer': 'Ella M. Charming', 'location': 'Berlin', 'title':
'Intermediate Python Training'}
```

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

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

```
>>> w.clear()
>>> print(w)
{}
```

UPDATE: MERGING DICTIONARIES

What about concatenating dictionaries, like we did with lists? There is something similar for dictionaries: the `update` method

`update()` merges the keys and values of one dictionary into another, overwriting values of the same key:

```
>>> knowledge = {"Frank": {"Perl"}, "Monica": {"C", "C++"}}
>>> knowledge2 = {"Guido": {"Python"}, "Frank": {"Perl", "Python"}}
```

```
>>> knowledge.update(knowledge2)
>>> knowledge
{'Frank': {'Python', 'Perl'}, 'Guido': {'Python'}, 'Monica': {'C', 'C++'}}
```

ITERATING OVER A DICTIONARY

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

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

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

```
>>> for key in d.keys():
...     print(key)
...
b
c
a
d
>>>
```

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

```
>>> for value in d.values():
...     print(value)
...
34
304
123
99
>>>
```

The above loop is logically equivalent to the following one:

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

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

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

```
In [5]: %%timeit d = {"a":123, "b":34, "c":304, "d":99}
for key in d.keys():
    x=d[key]
...
1000000 loops, best of 3: 225 ns per loop

In [6]: %%timeit d = {"a":123, "b":34, "c":304, "d":99}
for value in d.values():
    x=value
...
1000000 loops, best of 3: 164 ns per loop

In [7]:
```

CONNECTION BETWEEN LISTS AND DICTIONARIES

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

If we have a dictionary

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

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

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

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



LISTS FROM DICTIONARIES

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

```
>>> w = {"house":"Haus", "cat": "", "red": "rot"}
>>> items_view = w.items()
>>> items = list(items_view)
>>> items
[('house', 'Haus'), ('cat', ''), ('red', 'rot')]
>>>
>>> keys_view = w.keys()
>>> keys = list(keys_view)
>>> keys
['house', 'cat', 'red']
>>>
>>> values_view = w.values()
>>> values = list(values_view)
>>> values
['Haus', '', 'rot']
>>> values_view
dict_values(['Haus', '', 'rot'])
>>> items_view
dict_items([('house', 'Haus'), ('cat', ''), ('red', 'rot')])
>>> keys_view
dict_keys(['house', 'cat', 'red'])
>>>
```

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

TURN LISTS INTO DICTIONARIES

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

We have two lists, one containing the dishes and the other one the corresponding countries:

```
>>> dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA"]
```

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

```
>>> country_specialities_iterator = zip(countries, dishes)
>>> country_specialities_iterator
<zip object at 0x7fa5f7cad408>
>>> country_specialities = list(country_specialities_iterator)
>>> print(country_specialities)
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'),
 ('USA', 'hamburger')]
>>>
```

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

```
>>> country_specialities_dict = dict(country_specialities)
>>> print(country_specialities_dict)
{'USA': 'hamburger', 'Germany': 'sauerkraut', 'Spain': 'paella',
 'Italy': 'pizza'}
>>>
```

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

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

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

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

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

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

EVERYTHING IN ONE STEP

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

```
>>> country_specialities_dict = dict(list(zip(["pizza",
    "sauerkraut", "paella", "hamburger"], ["Italy", "Germany", "Spain",
    "USA", "Switzerland"])))
>>> print(country_specialities_dict)
{'paella': 'Spain', 'hamburger': 'USA', 'sauerkraut': 'Germany',
'pizza': 'Italy'}
>>>
```

On the other hand, the code in the previous script is gilding the lily:

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

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

DANGER LURKING

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

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

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

```
>>> l1 = ["a", "b", "c"]
>>> l2 = [1, 2, 3]
>>> c = zip(l1, l2)
```

```
>>> z1 = list(c)
>>> z2 = list(c)
>>> print(z1)
[('a', 1), ('b', 2), ('c', 3)]
>>> print(z2)
[]
```

As an exercise, you may muse about the following script.

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

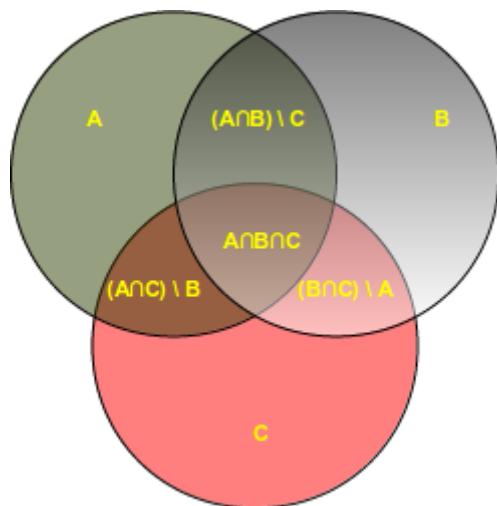
If you start this script, you will see that the dictionary you want to create will be empty:

```
$ python3 tricky_code.py
[('pizza', 'Italy'), ('sauerkraut', 'Germany'), ('paella', 'Spain'),
('hamburger', 'USA')]
{ }
$
```

SETS AND FROZENSETS

INTRODUCTION

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



Cantor defined a set at the beginning of his "Beiträge zur Begründung der transfiniten Mengenlehre":

"A set is a gathering together into a whole of definite, distinct objects of our perception and of our thought - which are called elements of the set." Nowadays, we can say in "plain" English: A set is a well defined collection of objects.

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

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

SETS

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

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

```
>>> x = set("A Python Tutorial")
>>> x
{'A', ' ', 'i', 'h', 'l', 'o', 'n', 'P', 'r', 'u', 't', 'a', 'y',
 'T'}
>>> type(x)
<class 'set'>
>>>
```

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

```
>>> x = set(["Perl", "Python", "Java"])
>>> x
{'Python', 'Java', 'Perl'}
>>>
```

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

```
>>> cities = set(("Paris", "Lyon",
 "London", "Berlin", "Paris", "Birmingham"))
>>> cities
{'Paris', 'Birmingham', 'Lyon', 'London', 'Berlin'}
>>>
```

As we have expected, no doublets are in the resulting set of cities.

IMMUTABLE SETS

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

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

FROZENSETS

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

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

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

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

IMPROVED NOTATION

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

```
>>> adjectives = {"cheap", "expensive", "inexpensive", "economical"}
>>> adjectives
{'inexpensive', 'cheap', 'expensive', 'economical'}
```

SET OPERATIONS

ADD(ELEMENT)

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

```
>>> colours = {"red", "green"}
>>> colours.add("yellow")
>>> colours
{'green', 'yellow', 'red'}
>>> colours.add(["black", "white"])
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

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

CLEAR()

All elements will be removed from a set.

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

COPY

Creates a shallow copy, which is returned.

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

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

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

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

DIFFERENCE()

This method returns the difference of two or more sets as a new set.

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

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

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

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

DIFFERENCE_UPDATE()

The method difference_update removes all elements of another set from this set.
`x.difference_update(y)` is the same as "`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
{'a', 'e', 'd'}
>>>
```

DISCARD(EL)

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

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

REMOVE(EL)

works like discard(), but if el is not a member of the set, a KeyError will be raised.

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

```

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

UNION(S)

This method returns the union of two sets as a new set, i.e. all elements that are in either set.

```

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

This can be abbreviated with the pipe operator "|":

```

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

INTERSECTION(S)

Returns the intersection of the instance set and the set s as a new set. In other words: A set with all the elements which are contained in both sets is returned.

```

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

This can be abbreviated with the ampersand operator "&":

```

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

ISDISJOINT()

This method returns True if two sets have a null 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()

`x.issubset(y)` returns True, if x is a subset of y. "`<=`" is an abbreviation for "Subset of" and "`>=`" for "superset of"

"`<`" is used to check if a set is a proper subset of a set.

```
>>> 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 proper subset of x
True
>>> x < x # a set can never be a proper subset of oneself.
False
>>> x <= x
True
>>>
```

ISSUPERSET()

`x.issuperset(y)` returns True, if x is a superset of y. "`>=`" is an abbreviation for "issuperset of"

"`>`" is used to check if a set is a proper superset of a set.

```
>>> 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()` removes and returns an arbitrary set element. The method raises a `KeyError` if the set is empty

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

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

AN EXTENSIVE EXAMPLE FOR SETS

PYTHON AND THE BEST NOVEL

This chapter deals with natural languages and literature. It will be also an extensive example and use case for Python sets. Novices in Python often think that sets are just a toy for mathematicians and that there is no real use case in programming. The contrary is true. There are multiple use cases for sets. They are used for example to get rid of doublets - multiple occurrences of elements - in a list, i.e. to make a list unique.



In the following example we will use sets to determine the different words occurring in a novel. Our use case is build around a novel which has been praised by many as the best novel in the English language and also as the hardest to read. We are talking about the novel "**Ulysses**" by James Joyce. We will not talk about or examine the beauty of the language or the language style. We will study the novel by having a close look at the words used in the novel. Our approach will be purely statically. The claim is that James Joyce used in his novel more words than any other author. Actually his vocabulary is above and beyond all other authors, maybe even Shakespeare.

Besides Ulysses we will use the novels "**Sons and Lovers**" by D.H. Lawrence, "**The Way of All Flesh**" by Samuel Butler, "**Robinson Crusoe**" by Daniel Defoe, "**To the Lighthouse**" by Virginia Woolf, "**Moby Dick**" by Herman Melville and the Short Story "**Metamorphosis**" by Franz Kafka.

Before you continue with this chapter of our tutorial it might be a good idea to read the chapter **Sets and Frozen Sets** and the two chapter on **regular expressions** and **advanced regular expressions**.

DIFFERENT WORDS OF A TEXT

To cut out all the words of the novel "Ulysses" we can use the function `findall` from the module "re":

```
import re
# we don't care about case sensitivity and therefore use lower:
```

```
ulysses_txt = open("books/james_joyce_ulysses.txt").read().lower()
words = re.findall(r"\b[\w-]+\b", ulysses_txt)
print("The novel ulysses contains " + str(len(words)))
```

The novel ulysses contains 272452

This number is the sum of all the words and many words occur multiple time:

```
for word in ["the", "while", "good", "bad", "ireland", "irish"]:
    print("The word '" + word + "' occurs " + \
          str(words.count(word)) + " times in the novel!" )
```

```
The word 'the' occurs 15112 times in the novel!
The word 'while' occurs 123 times in the novel!
The word 'good' occurs 321 times in the novel!
The word 'bad' occurs 90 times in the novel!
The word 'ireland' occurs 90 times in the novel!
The word 'irish' occurs 117 times in the novel!
```

272452 surely is a huge number of words for a novel, but on the other hand there are lots of novels with even more words. More interesting and saying more about the quality of a novel is the number of different words. This is the moment where we will finally need "set". We will turn the list of words "words" into a set. Applying "len" to this set will give us the number of different words:

```
diff_words = set(words)
print("'Ulysses' contains " + str(len(diff_words)) + " different
words!")
```

'Ulysses' contains 29422 different words!

This is indeed an impressive number. You can see this, if you look at the other novels in our folder books:

```
novels = ['sons_and_lovers_lawrence.txt',
          'metamorphosis_kafka.txt',
          'the_way_of_all_flash_butler.txt',
          'robinson_crusoe_defoe.txt',
          'to_the_lighthouse_woolf.txt',
          'james_joyce_ulysses.txt',
          'moby_dick_melville.txt']
for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    diff_words = set(words)
    n = len(diff_words)
    print("{name:38s}: {n:5d}".format(name=novel[:-4], n=n))

sons_and_lovers_lawrence : 10822
metamorphosis_kafka : 3027
the_way_of_all_flash_butler : 11434
robinson_crusoe_defoe : 6595
```

to_the_lighthouse_woolf	:	11415
james_joyce_ulysses	:	29422
moby_dick_melville	:	18922

SPECIAL WORDS IN ULYSSES

We will subtract all the words occurring in the other novels from "Ulysses" in the following little Python program. It is amazing how many words are used by James Joyce and by none of the other authors:

```

words_in_novel = {}
for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words

words_only_in_ulysses =
set(words_in_novel['james_joyce_ulysses.txt'])
novels.remove('james_joyce_ulysses.txt')
for novel in novels:
    words_only_in_ulysses -= set(words_in_novel[novel])

with open("books/words_only_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_only_in_ulysses)
    fh.write(txt)

print(len(words_only_in_ulysses))

```

15314

By the way, Dr. Seuss wrote a book with only 50 different words: Green Eggs and Ham

The **file with the words only occurring in Ulysses** contains strange or seldom used words like:

huntingcrop tramtrack pappin kithogue pennyweight undergarments scission nagyaságos
wheedling begad dogwhip hawthornden turnbull calumet covey repudiated pendennis
waistcoatpocket nostrum

COMMON WORDS

It is also possible to find the words which occur in every book. To accomplish this, we need the set intersection:

```

# we start with the words in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

```

```
print(len(common_words))
```

1745

DOING IT RIGHT

We made a slight mistake in the previous calculations. If you look at the texts, you will notice that they have a header and footer part added by Project Gutenberg, which doesn't belong to the texts. The texts are positioned between the lines:

START OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

and

END OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

or

*** START OF THIS PROJECT GUTENBERG EBOOK ULYSSES ***

and

*** END OF THIS PROJECT GUTENBERG EBOOK ULYSSES ***

The function `read_text` takes care of this:

```
def read_text(fname):
    beg_e = re.compile(r"\*\*\* ?start of (this|the) project
gutenberg ebook[\*\*]*\*\*\*")
    end_e = re.compile(r"\*\*\* ?end of (this|the) project gutenberg
ebook[\*\*]*\*\*\*")
    txt = open("books/" + fname).read().lower()
    beg = beg_e.search(txt).end()
    end = end_e.search(txt).start()
    return txt[beg:end]
words_in_novel = {}
for novel in novels + ['james_joyce_ulysses.txt']:
    txt = read_text(novel)
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words
words_in_ulysses = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    words_in_ulysses -= set(words_in_novel[novel])

with open("books/words_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_in_ulysses)
    fh.write(txt)

print(len(words_in_ulysses))
```

15341

```
# we start with the words in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

print(len(common_words))
```

1279

The words of the set "common_words" are words belong to the most frequently used words of the English language. Let's have a look at 30 arbitrary words of this set:

```
counter = 0
for word in common_words:
    print(word, end=", ")
    counter += 1
    if counter == 30:
        break

ancient, broke, breathing, laugh, divided, forced, wealth, ring,
outside, throw, person, spend, better, errand, school, sought,
knock, tell, inner, run, packed, another, since, touched, bearing,
repeated, bitter, experienced, often, one,
```

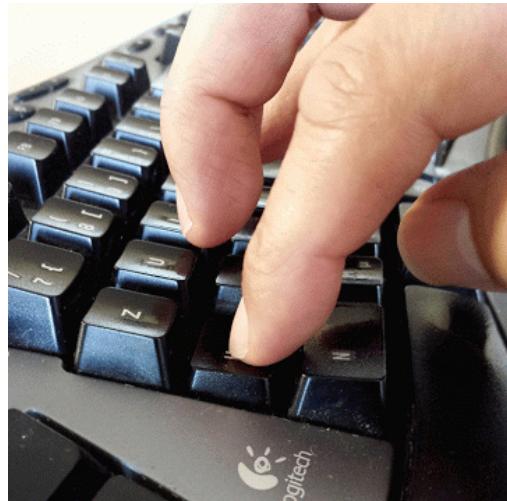
© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

INPUT FROM KEYBOARD

THE INPUT FUNCTION

There are hardly any programs without any input. Input can come in various ways, for example, from a database, another computer, mouse clicks and movements or from the internet. Yet, in most cases the input stems from the keyboard. For this purpose, Python provides the function `input()`. `input` has an optional parameter, which is the prompt string.

If the `input` function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key. The text of the optional parameter, i.e. the prompt, will be printed on the screen.



The input of the user will be returned as a string without any changes. If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the `eval` function.

Let's have a look at the following example:

```
name = input("What's your name? ")
print("Nice to meet you " + name + "!")
age = input("Your age? ")
print("So, you are already " + age + " years old, " + name + "!")
```

We save the program as "input_test.py" and run it:

```
$ python input_test.py
What's your name? "Frank"
Nice to meet you Frank!
Your age? 42
So, you are already 42 years old, Frank!
```

We will further experiment with the `input` function in the following interactive Python session:

```
>>> cities_canada = input("Largest cities in Canada: ")
Largest cities in Canada: ["Toronto", "Montreal", "Calgara",
"Ottawa"]
>>> print(cities_canada, type(cities_canada))
```

```

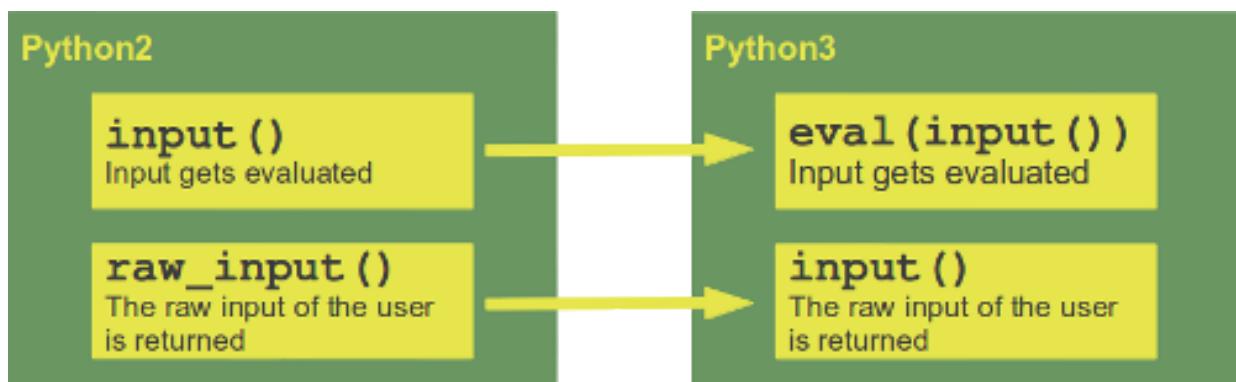
["Toronto", "Montreal", "Calgara", "Ottawa"] <class 'str'>
>>>
>>> cities_canada = eval(input("Largest cities in Canada: "))
Largest cities in Canada: ["Toronto", "Montreal", "Calgara",
"Ottawa"]
>>> print(cities_canada, type(cities_canada))
['Toronto', 'Montreal', 'Calgara', 'Ottawa'] <class 'list'>
>>>
>>> population = input("Population of Toronto? ")
Population of Toronto? 2615069
>>> print(population, type(population))
2615069 <class 'str'>
>>>
>>> population = int(input("Population of Toronto? "))
Population of Toronto? 2615069
>>> print(population, type(population))
2615069 <class 'int'>
>>>

```

DIFFERENCES TO PYTHON2

The usage of `input` or better the implicit evaluation of the input has often lead to serious programming mistakes in the earlier Python versions, i.e. 2.x Therefore, there the `input` function behaves like the `raw_input` function from Python2.

The changes between the versions are illustrated in the following diagram:



CONDITIONAL STATEMENTS

WHAT IF THERE WERE NO DECISIONS TO BE MADE?

Our subtitle is a purely rhetorical question, because there are always decisions to be made, both in real life and of course in programming. Under certain conditions some decisions are inevitable in normal life, e.g. man and women just have to go separate ways from time to time, as the image indicates.

Conditionals, - mostly in the form of if statements - are one of the essential features of a programming language and Python is no exception. You will find hardly any programming language without an if statement.¹ There is hardly a way to program without having branches in the flow of code. At least, if the code has to solve some useful problem.



A decision has to be taken when the script or program comes to a point where it has a choice of actions, i.e. different computations, to choose from.

The decision depends in most cases on the value of variables or arithmetic expressions. These expressions are evaluated to the Boolean values True or False. The statements for the decision taking are called conditional statements. Alternatively they are also known as conditional expressions or conditional constructs.

So this chapter deals with conditionals. But to code them in Python, we have to know how to combine statements into a block. It also seems to be an ideal moment to introduce the Python block principle in combination with conditional statements.

BLOCKS AND INDENTATIONS

The concept of block building is also known in natural languages, as we can deduce from the following text:

If it rains tomorrow, I will tidy up the cellar. After this I will paint the walls. If there is some time left, I will do my tax declaration.

As we all know, there will be no time left to deal with the tax declaration. Joking apart, we can see a sequence of actions in the previous text, which have to be performed in a

chronological order. If you have a closer look at the text, you will notice that it is ambiguous: Is the action of painting the walls also linked to the event of rain? Especially, doing the tax declaration, does it depend on the rain as well? What will this person do, if it doesn't rain? We extend the text by creating further ambiguities:

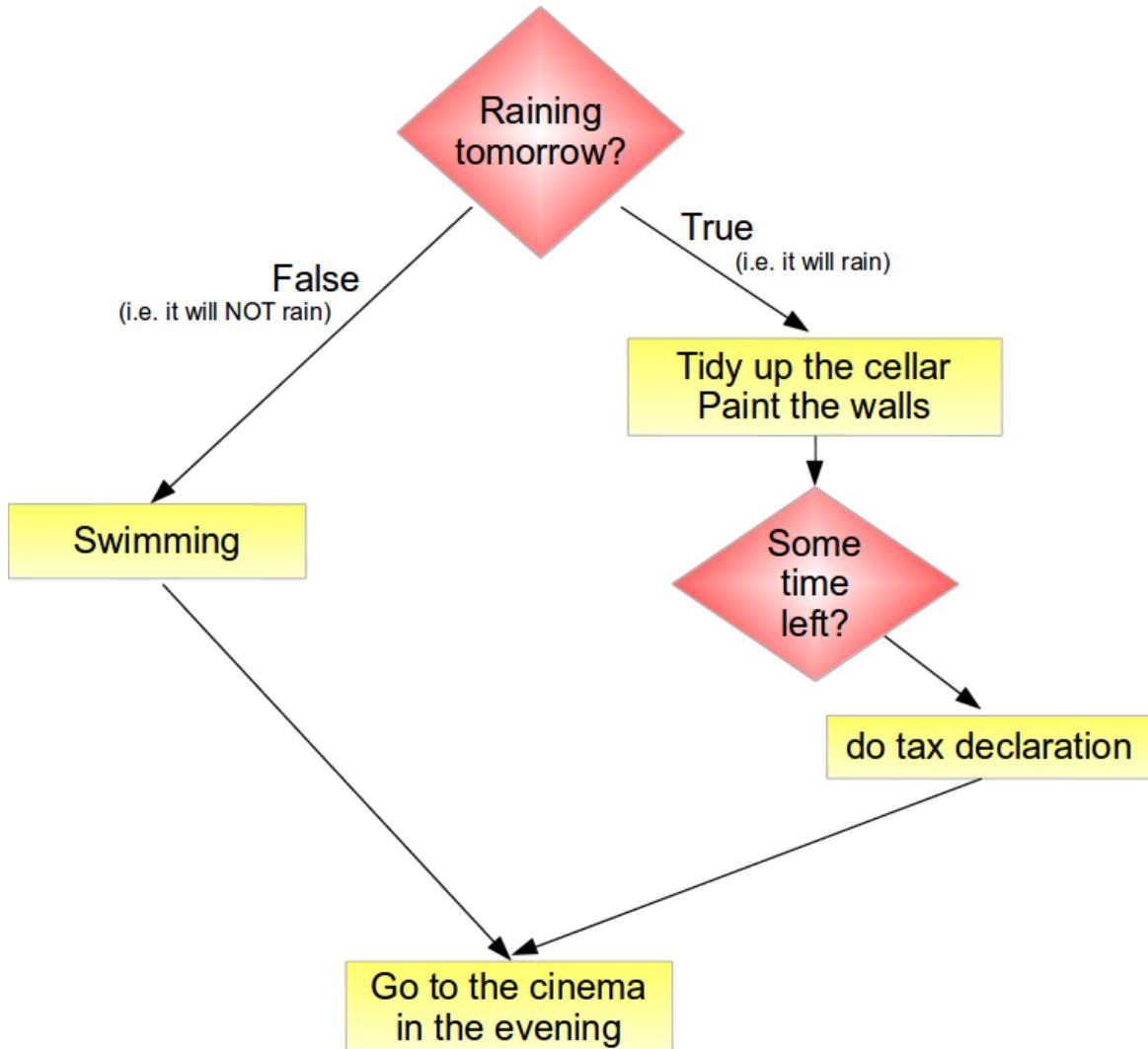
If it rains tomorrow, I will tidy up the cellar. After this I will paint the walls. If there is some time left, I will do my tax declaration. Otherwise, I will go swimming. In the evening, I will go to the cinema with my wife!

Can his wife hope to be invited to the cinema? Does she have to pray or hope for rain? To disambiguate the text we can phrase it in a way, which is closer to programming code and Python code and hopefully, a happy ending for his wife:

```
If it rains tomorrow, I will do the following:
- tidy up the cellar
- paint the walls
- If there is some time left, I will
    - do my tax declaration
Otherwise, I will do the following:
- go swimming
go to the cinema with my wife in the evening
```

Such a work flow is often formulated in the programming environment as a so-called flow chart or programming flowchart:





Blocks are used in programming to enable groups of statements to be seen or treated as if they were one statement. A block consists of one or more statements. A program can be seen as a block, which consists of statements and other nested blocks. There have been different approaches in programming languages to syntactically describe blocks: ALGOL 60 and Pascal, for example, use "begin" and "end", while C and similar languages use curly braces "{" and "}". Bash has yet another design by using do ... done and if ... fi or case ... esac constructs.

There is a big disadvantage for all of these approaches: The code may be all right for the interpreter or the compiler of the language, but it can be written in a way, which is badly structured for humans. We want to illustrate this in the following "pseudo" C-Code snippet:

```

if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
        do_taxes();
} else
  
```

```

        enjoy_swimming();
go_cinema();

```

Let us include some blanks in front of the go_cinema call:

```

if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
        do_taxes();
} else
    enjoy_swimming();
    go_cinema();

```

The execution of the program will not change: They will go to the cinema whether it rains or not! The way the code is written misleadingly connotes the meaning that they will only go to cinema, if it does not rain. This example demonstrates the possible ambiguity in interpreting C-Code by humans. This means, that people write code, which will lead to a program which will not "behave" in the intended way.

There is even another danger lurking in this example. What if the programmer has forgotten to include the last two statements into braces? Should the code be like this:

```

if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
        do_taxes();
} else {
    enjoy_swimming();
    go_cinema();
}

```

Now, they will only go to cinema, if it is not a rainy day, which makes sense, if it is an open air cinema. The following code is the correct code, if they want to go to the cinema regardless of the weather:

```

if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
        do_taxes();
} else {
    enjoy_swimming();
}
go_cinema(); }

```

The problem in C is that the way we indent code has no meaning for the compiler and it might suggest the wrong interpretation for humans, if they try to understand the code.

This is different in Python. Blocks are created with indentations. We could say "What you see is what you get!"

The example above may look in a Python program like this:

```

if raining_tomorrow:
    tidy_up_the_cellar()
    paint_the_walls()
    if time_left:
        do_taxes()
else:
    enjoy_swimming()
go_cinema()

```

There is no ambiguity in the Python version. The couple's visit to the cinema is bound to occur, regardless of the weather. Moving the `go_cinema()` call to the same indentation level as the `enjoy_swimming()`, changes a lot. In this case there will be no cinema, if it rains.

We have just seen that it is useful to use indentation in C or C++ programs to increase or ensure the legibility of a program for the programmer but not for the C compiler. The Compiler relies solely on the structuring determined by the braces. Code in Python can only and has to be structured by using the correct indentation. This means that Python forces the programmer to use the indentation that he or she is supposed to use anyway to write nice code. So, Python does not allow to obfuscate the structure of a program by using bogus or misleading indentations.

A block of code in Python has to be indented by the same amount of blanks or tabs. We will further deepen this in the next subsection on the conditional statements in Python.

CONDITIONAL STATEMENTS IN PYTHON

As we have already stated, the `if`-statements are used to change the flow of control in a Python program. This makes it possible to decide at run-time whether or not to run one block of code or another.

The simplest form of an `if` statement in Python looks like this:

```

if condition:
    statement
    statement
    # ... some more indented statements if necessary

```

The indented block of code is executed only if the condition "condition" is evaluated to True, meaning that it is logically true.

The following program code asks the user about his or her nationality. The indented `print` statement will only be executed, if the nationality is "French". If the user of this program uses another nationality, nothing will be printed:

```

person = input("Nationality? ")
if person == "french":

```

```
print("Préférez-vous parler français?")
```

Please note that if somebody types in "French", nothing will be printed either, because we solely check the lower case spelling. We can change this by extending the condition with an "or":

```
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
```

Your Italian colleague may protest that Italian speakers will not be taken into consideration by our previous little program. We can change this by adding another if:

```
person = input("Nationality? ")
if person == "french" or person == "French" :
    print("Préférez-vous parler français?")
if person == "italian" or person == "Italian" :
    print("Preferisci parlare italiano?")
```

This small python script has a disadvantage: Let's assume that someone inputs "french" as a nationality. In this case, the print below the first "if" will be executed, i.e. the text "Préférez-vous parler français?" will be printed. After this the program will check, if the value for person is equal to "italian" and "Italian", which cannot be the case, as we assumed "french" is the input. This means that our program performs an unnecessary test, if the input is "french" or "French".

This problem can be solved with an "elif" condition. The expression is only checked after "elif", if the expression in the previous "elif" or "if" was "false".

```
person = input("Nationality? ")
if person == "french" or person == "French" :
    print("Préférez-vous parler français?")
elif person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
else:
    print("You are neither Italian nor French,")
    print("so we have to speak English with each other.")
```

Like in our previous example, if statements have in many cases "elif" and "else" branches as well. To be precise: There can be more than one "elif" branch, but only one "else" branch. The else branch has to be at the end of the if statement, i.e. it can't be followed by other elif branches.

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

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
```

```
...
elif another_condition:
    another_statement_block
else:
    else_block
```

If the condition "condition_1" is True, the statements of the block statement_block_1 will be executed. If not, condition_2 will be evaluated. If condition_2 evaluates to True, statement_block_2 will be executed, if condition_2 is False, the other conditions of the following elif conditions will be checked, and finally if none of them has been evaluated to True, the indented block below the else keyword will be executed.

EXAMPLE: DOG YEARS

It's a generally accepted belief, to assume that one year in the life of a dog corresponds to seven years in the life of a human being. But apparently there are other more subtle methods to calculate this haunting problem, haunting at least for some dog owners. Another subtler - and some think a preciser method - works like this:

- A one year old dog roughly corresponds to a fourteen year old child
- A dog who is two years old corresponds to a 22 year old human
- Every further dog year corresponds to five human years

The following Python script implements this "dog years algorithm":

```
age = int(input("Age of the dog: "))
print()
if age < 1:
    print("This can hardly be true!")
elif age == 1:
    print("about 14 human years")
elif age == 2:
    print("about 22 human years")
elif age > 2:
    human = 22 + (age -2)*5
    print("Human years: ", human)

###
```

There is one drawback to the script. It works only for integers, i.e. full years.

TRUE OR FALSE

Unfortunately it is not as easy in real life as it is in Python to differentiate between true and false:

The following objects are evaluated by Python as False:

- numerical zero values (0, 0.0, 0.0+0.0j),
- the Boolean value False,
- empty strings,
- empty lists and empty tuples,
- empty dictionaries.
- plus the special value None.

All other values are considered to be True.

THE TERNARY IF

C programmers usually know the following abbreviated notation for the if construct:

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

This is an abbreviation for the following C code:

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

C programmers have to get used to a different notation in Python:

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

The Python version is by far more readable. The expression above, can be read as "max shall be a if a is greater than b else b".

But the ternary if statement is more than an abbreviation. It is an expression, which can be used within another expression:

```
max = (a if (a > b) else b) * 2.45 - 4
```

Using if statements within programs can easily lead to complex decision trees, i.e. every if statements can be seen like the branches of a tree.

We will read in three float numbers in the following program and will print out the largest

value:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y and x > z:
    maximum = x
elif y > x and y > z:
    maximum = y
else:
    maximum = z

print("The maximal value is: " + str(maximum))
```

There are other ways to write the conditions like the following one:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

if x > y:
    if x > z:
        maximum = x
    else:
        maximum = z
else:
    if y > z:
        maximum = y
    else:
        maximum = z

print("The maximal value is: " + str(maximum))
```

Another way - which is less efficient, because we have to create a tuple or list to compare the numbers - to do it, can be seen in the following example. We are using the built-in function max, which calculates the maximum of a list or a tuple:

```
x = float(input("1st Number: "))
y = float(input("2nd Number: "))
z = float(input("3rd Number: "))

maximum = max((x,y,z))

print("The maximal value is: " + str(maximum))
```

If we want to read in an arbitrary number of elements, which we do not want or need to save in a list, we can calculate the maximum in the following way:

```
number_of_values = int(input("How many values? "))
```

```
maximum = float(input("Value: "))
for i in range(number_of_values - 1):
    value = float(input("Value: "))
    if value > maximum:
        maximum = value

print("The maximal value is: " + str(maximum))
```

FOOTNOTES:

¹ LOOP is a programming language without conditionals, but this language is purely pedagogical. It has been designed by the German computer scientist Uwe Schöning. The only operations which are supported by LOOP are assignments, additions and loopings. But LOOP is of no practical interest and besides this it is only a proper subset of the computable functions.

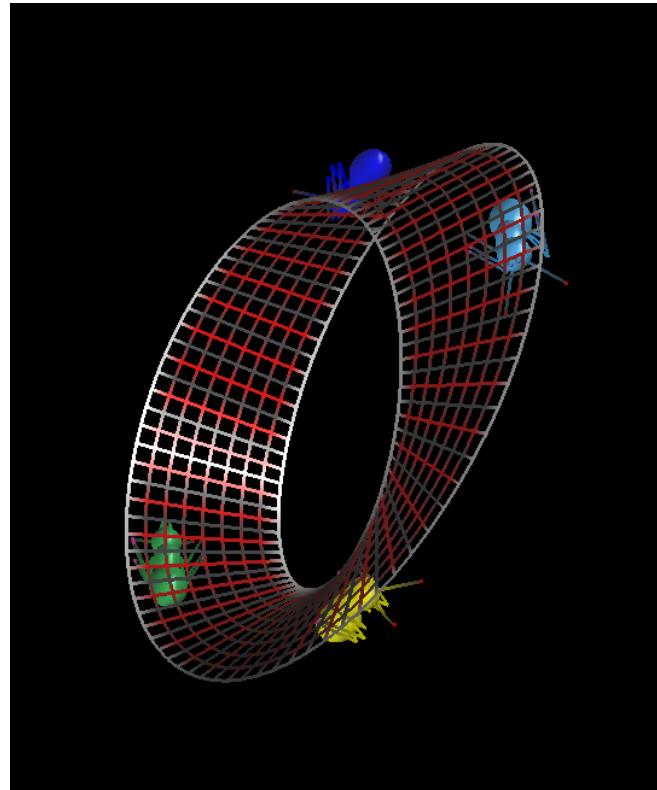
LOOPS

GENERAL STRUCTURE OF A LOOP

Many algorithms make it necessary for a programming language to have a construct which makes it possible to carry out a sequence of statements repeatedly. The code within the loop, i.e. the code carried out repeatedly, is called the body of the loop.

Essentially, there are three different kinds of loops:

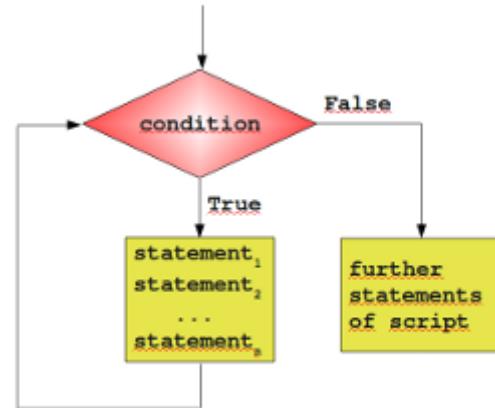
- Count-controlled loops
A construction for repeating a loop a certain number of times. An example of this kind of loop is the for-loop of the programming language C:
`for (i=0; i <= n; i++)`
Python doesn't know this kind of loop.
- Condition-controlled loop
A loop will be repeated until a given condition changes, i.e. changes from True to False or from False to True, depending on the kind of loop. There are while loops and do while loops with this behaviour.
- Collection-controlled loop
This is a special construct which allow looping through the elements of a "collection", which can be an array, list or other ordered sequence. Like the for loop of the bash shell (e.g. `for i in *`, `do echo $i; done`) or the foreach loop of Perl.



Python supplies two different kinds of loops: the while loop and the for loop, which correspond to the condition-controlled loop and collection-controlled loop.

Most loops contain a counter or more generally variables, which change their values in the course of calculation. These variables have to be initialized before the loop is started. The counter or other variables, which can be altered in the body of the loop, are contained in the condition. Before the body of the loop is executed, the condition is evaluated. If it evaluates to False, the while loop is finished. This means that the program flow will continue with the first statement after the while statement, i.e. on the same indentation level as the while loop.

If the condition is evaluated to True, the body,
 - the indented block below the line with
 "while" - gets executed. After the body is
 finished, the condition will be evaluated again.
 The body of the loop will be executed as long
 as the condition yields True.



A SIMPLE EXAMPLE WITH A WHILE LOOP

It's best to illustrate the operating principle of a loop with a simple Python example. The following small script calculates the sum of the numbers from 1 to 100. We will later introduce a more elegant way to do it.

```

#!/usr/bin/env python3

n = 100

s = 0
counter = 1
while counter <= n:
    s = s + counter
    counter += 1

print("Sum of 1 until %d: %d" % (n,s))
  
```

USING A WHILE LOOP FOR READING STANDARD INPUT

Before we go on with the while loop, we want to introduce some fundamental things on standard input and standard output. Normally, the keyboard serves as the standard input. The standard output is usually the terminal or console where the script had been started, which prints the output. A script is supposed to send its error messages to the standard error.

Python has these three channels as well:

- standard input
- standard output
- standard error

They are contained in the module sys. Their names are:

- sys.stdin
- sys.stdout

- sys.stderr

The following script shows how to read with a while loop character by character from standard input (keyboard).

```
import sys

text = ""
while 1:
    c = sys.stdin.read(1)
    text = text + c
    if c == '\n':
        break

print("Input: %s" % text)
```

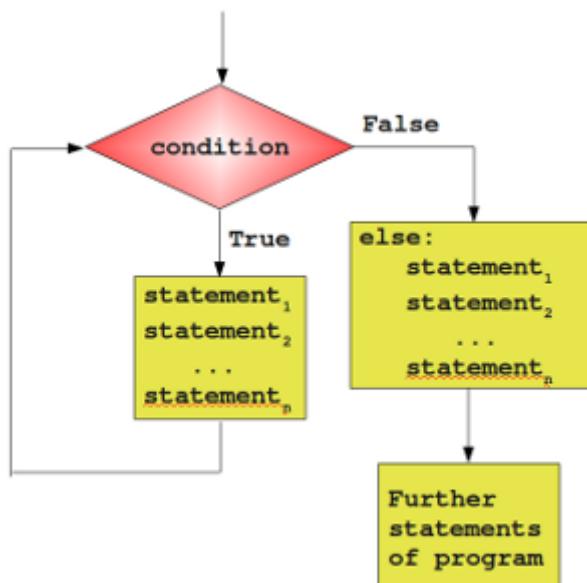
Though it's possible to read input like this, usually the function `input()` is used.

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

THE ELSE PART

Similar to the `if` statement, the while loop of Python has also an optional `else` part. This is an unfamiliar construct for many programmers of traditional programming languages.

The statements in the `else` part are executed, when the condition is not fulfilled anymore. Some might ask themselves now, where the possible benefit of this extra branch is. If the statements of the additional `else` part were placed right after the while loop without an `else`, they would have been executed anyway, wouldn't they. We need to understand a new language construct, i.e. the `break` statement, to obtain a benefit from the additional `else` branch of the while loop. The general syntax of a while loop looks like this:



```

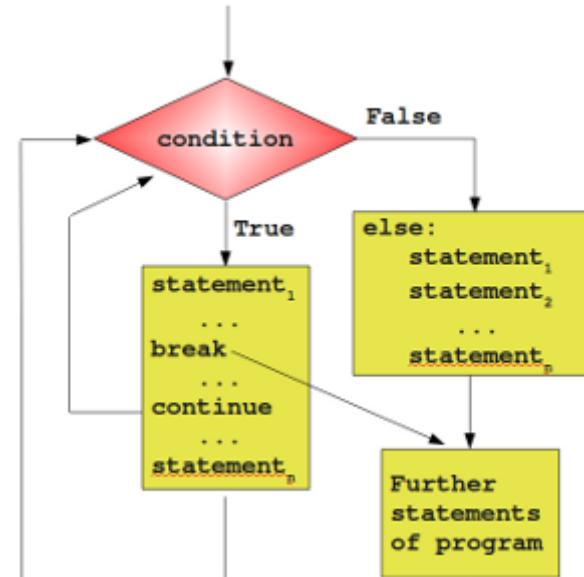
while condition:
    statement_1
    ...
    statement_n
else:
    statement_1
    ...
    statement_n

```

PREMATURE TERMINATION OF A WHILE LOOP

So far, a while loop only ends, if the condition in the loop head is fulfilled. With the help of a break statement a while loop can be left prematurely, i.e. as soon as the control flow of the program comes to a break inside of a while loop (or other loops) the loop will be immediately left. "break" shouldn't be confused with the continue statement. "continue" stops the current iteration of the loop and starts the next iteration by checking the condition.

Now comes the crucial point: If a loop is left by break, the else part is not executed.



This behaviour will be illustrated in the following example, a little guessing number game. A human player has to guess a number between a range of 1 to n. The player inputs his guess. The program informs the player, if this number is larger, smaller or equal to the secret number, i.e. the number which the program has randomly created. If the player wants to give up, he or she can input a 0 or a negative number.

Hint: The program needs to create a random number. Therefore it's necessary to include the module "random".

```

import random
n = 20
to_be_guessed = int(n * random.random()) + 1
guess = 0
while guess != to_be_guessed:
    guess = int(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("Sorry that you're giving up!")
        break

```

```
else:  
    print("Congratulation. You made it!")
```

The output of a game session might look like this:

```
$ python3 number_game.py  
New number: 12  
Number too small  
New number: 15  
Number too small  
New number: 18  
Number too large  
New number: 17  
Congratulation. You made it!  
$
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

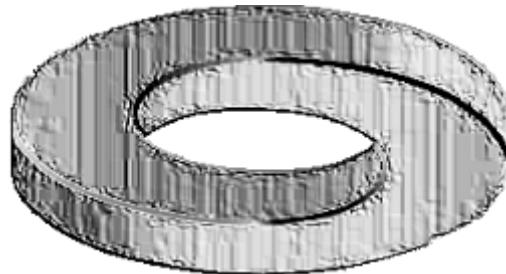
FOR LOOPS

INTRODUCTION

Like the while loop the for loop is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times.

There are hardly programming languages without for loops, but the for loop exists in many different flavours, i.e. both the syntax and the semantics differs from one programming language to another.

Different kinds of for loops:



- Count-controlled for loop (Three-expression for loop)

This is by far the most common type. This statement is the one used by C. The header of this kind of for loop consists of a three-parameter loop control expression. Generally it has the form:

```
for (A; Z; I)
```

A is the initialisation part, Z determines a termination expression and I is the counting expression, where the loop variable is incremented or decremented. An example of this kind of loop is the for-loop of the programming language C:

```
for (i=0; i <= n; i++)
```

This kind of for loop is not implemented in Python!

- Numeric Ranges

This kind of for loop is a simplification of the previous kind. It's a counting or enumerating loop. Starting with a start value and counting up to an end value, like `for i = 1 to 100`

Python doesn't use this either.

- Vectorized for loops

They behave as if all iterations are executed in parallel. This means, for example, that all expressions on the right side of assignment statements get evaluated before the assignments.

- Iterator-based for loop

Finally, we come to the one used by Python. This kind of a for loop iterates over an enumeration of a set of items. It is usually characterized by the use of an implicit or explicit iterator. In each iteration step a loop variable is set to a value in a sequence or other data collection. This kind of for loop is known in most Unix and Linux shells and it is the one which is implemented in Python.

SYNTAX OF THE FOR LOOP

As we mentioned earlier, the Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

The items of the sequence object are assigned one after the other to the loop variable; to be precise the variable points to the items. For each item the loop body is executed.

Example of a simple for loop in Python:

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

The else block is special; while Perl programmer are familiar with it, it's an unknown concept to C and C++ programmers. Semantically, it works exactly as the optional else of a while loop. It will be executed only if the loop hasn't been "broken" by a break statement. So it will only be executed, after all the items of the sequence in the header have been used.



If a break statement has to be executed in the program flow of the for loop, the loop will be exited and the program flow will continue with the first statement following the for loop, if there is any at all. Usually break statements are wrapped into conditional statements, e.g.

```
edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
```

```

print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")

```

If we call this script, we get the following result:

```

$ python for.py
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
$ 

```

Removing "spam" from our list of edibles, we will gain the following output:

```

$ python for.py
Great, delicious ham
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
$ 

```

Maybe, our disgust with spam is not so high that we want to stop consuming the other food. Now, this calls into play the continue statement. In the following little script we use the continue statement to go on with our list of edibles, when we have encountered a spam item. So continue prevents us from eating spam!

```

edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)
    # here can be the code for enjoying our food :-)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")

```

The output looks as follows:

```

$ python for.py
Great, delicious ham
No more spam please!
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
$ 

```

THE RANGE() FUNCTION

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions:

Example:

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

`range(n)` generates an iterator to progress the integer numbers starting with 0 and ending with $(n - 1)$. To produce the list with these numbers, we have to cast `range()` with the `list()`, as we did in the previous example.

`range()` can also be called with two arguments:

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

The above call produces the list iterator of numbers starting with `begin` (inclusive) and ending with one less than the number "end".

Example:

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

So far the increment of `range()` has been 1. We can specify a different increment with a third argument. The increment is called the "step". It can be both negative and positive, but not zero:

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

Example with step:

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

It can be done backwards as well:

```
>>> list(range(42,-12,-7))
[42, 35, 28, 21, 14, 7, 0, -7]
>>>
```

The `range()` function is especially useful in combination with the `for` loop, as we can see in the following example. The `range()` function supplies the numbers from 1 to 100 for the `for` loop to calculate the sum of these numbers:

```
n = 100
sum = 0
for counter in range(1,n+1):
    sum = sum + counter

print("Sum of 1 until %d: %d" % (n,sum))
```

CALCULATION OF THE PYTHAGOREAN NUMBERS

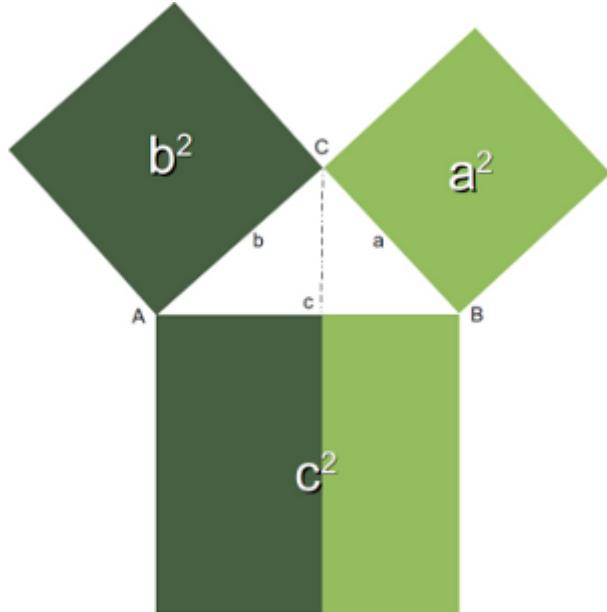
Generally, it is assumed that the Pythagorean theorem was discovered by Pythagoras that is why it has its name. But there is a debate whether the Pythagorean theorem might have been discovered earlier or by others independently. For the Pythagoreans, - a mystical movement, based on mathematics, religion and philosophy, - the integer numbers satisfying the theorem were special numbers, which had been sacred to them.

These days Pythagorean numbers are not mystical anymore. Though to some pupils at school or other people, who are not on good terms with mathematics, they may still appear so.

So the definition is very simple:
Three integers satisfying $a^2+b^2=c^2$ are called Pythagorean numbers.

The following program calculates all pythagorean numbers less than a maximal number.
Remark: We have to import the math module to be able to calculate the square root of a number.

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



ITERATING OVER LISTS WITH RANGE()

If you have to access the indices of a list, it doesn't look to be a good idea to use the for loop to iterate over the lists. We can access all the elements, but the index of an element is not available. But there is a way to access both the index of an element and the element itself. The solution consists in using range() in combination with the length function len():

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

```

        print(i,fibonacci[i])
print()

```

The output looks like this:

```

0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21

```

Remark: If you apply `len()` to a list or a tuple, you get the number of elements of this sequence.

LIST ITERATION WITH SIDE EFFECTS

If you loop over a list, it's best to avoid changing the list in the loop body. To give you an example, what can happen, have a look at the following example:

```

colours = ["red"]
for i in colours:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)

```

What will be printed by `"print(colours)"`?

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

To avoid these side effects, it's best to work on a copy by using the slicing operator, as can be seen in the next example:

```

colours = ["red"]
for i in colours[:]:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)

```

Now the output looks like this:

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

We still might have done something, what we shouldn't have done. We changed the list "colours", but our change hasn't had any effect on the loop anymore. The elements to be

looped remained the same during the iterations.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

ITERATORS AND ITERABLES

The Python forums and other question-and-answer websites like Quora and Stackoverflow are full of questions concerning 'iterators' and 'iterable'. Some want to know how they are defined and others want to know if there is an easy way to check, if an object is an iterator or an iterable. We will provide further down a function for this purpose.

We have seen that we can loop or iterate over various Python objects like lists, tuples and strings for example.



```
for city in ["Berlin", "Vienna", "Zurich"]:  
    print(city)  
for city in ("Python", "Perl", "Ruby"):  
    print(city)  
for char in "Iteration is easy":  
    print(char)
```

```
Berlin  
Vienna  
Zurich  
Python  
Perl  
Ruby  
I  
t  
e  
r  
a  
t  
i  
o  
n  
  
i  
s  
  
e  
a  
s  
y
```

This form of looping can be seen as iteration. Iteration is not restricted to explicit for loops. If you call the function sum, - e.g. on a list of integer values, - you do iteration as well.

So what is the difference between an **iterable** and an **iterator**?

In one perspective they are the same: You can iterate with a for loop over iterators and iterables. Every iterator is also an iterable, but not every iterable is an iterator. E.g. a list is iterable but a list is not an iterator! An iterator can be created from an iterable by using the function 'iter'. To make this possible the class of an object needs either a method '`__iter__`', which returns an iterator, or a '`__getitem__`' method with sequential indexes starting with 0.

Iterators are objects with a '`__next__`' method, which will be used when the function 'next' is called.

So what is going on behind the scenes, when a for loop is executed? The for statement calls `iter()` on the object (which should be a so-called container object), which it is supposed to loop over. If this call is successful, the `iter` call will return return an iterator object that defines the method `__next__()` which accesses elements of the object one at a time. The `__next__()` method will raise a `StopIteration` exception, if there are no further elements available. The for loop will terminate as soon as it catches a `StopIteration` exception. You can call the `__next__()` method using the `next()` built-in function. This is how it works:

```
cities = ["Berlin", "Vienna", "Zurich"]
iterator_obj = iter(cities)
print(iterator_obj)
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))

<list_iterator object at 0x7f08a055e0b8>
Berlin
Vienna
Zurich
```

If we called `next(iterator_obj)` one more time, it would return 'StopIteration'

The following function '`iterable`' will return True, if the object '`obj`' is an iterable and False otherwise.

```
def iterable(obj):
    try:
        iter(obj)
        return True
    except TypeError:
        return False

for element in [34, [4, 5], (4, 5), {"a":4}, "dfsdf", 4.5]:
    print(element, "iterable: ", iterable(element))

34 iterable:  False
[4, 5] iterable:  True
(4, 5) iterable:  True
{'a': 4} iterable:  True
dfsdf iterable:  True
4.5 iterable:  False
```

We have described how an iterator works. So if you want to add an iterator behavior to your class, you have to add the `__iter__` and the `__next__` method to your class. The `__iter__` method returns an iterator object. If the class contains a `__next__`, it is enough for the `__iter__` method to return `self`, i.e. a reference to itself:

```
class Reverse:
    """
    Creates Iterators for looping over a sequence backwards.
    """

    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
lst = [34, 978, 42]
lst_backwards = Reverse(lst)
for el in lst_backwards:
    print(el)
```

42
978
34

PRINT

INTRODUCTION

In principle, every computer program has to communicate with the environment or the "outside world". To this purpose nearly every programming language has special I/O functionalities, i.e. input/output. This ensures the interaction or communication with other components e.g. a database or a user. Input often comes - as we have already seen - from the keyboard and the corresponding Python command or better the corresponding Python function for reading from the standard input is `input()`.

We have also seen in previous examples of our tutorial that we can write into the standard output by using `print`. In this chapter of our tutorial we want to have a detailed look at the `print` function. As some might have skipped over it, we want to emphasize that we wrote "`print function`" and not "`print statement`". You can easily find out how crucial this difference is, if you take an arbitrary Python program written in version 2.x and if you try to let it run with a Python3 interpreter. In most cases you will receive error messages.

One of the most frequently occurring errors will be related to `print`, because most programs contain prints. We can generate the most typical error in the interactive Python shell:



```
$ 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 42
  File "", line 1
    print 42
          ^
SyntaxError: invalid syntax
>>>
```

This is a familiar error message for most of us: We have forgotten the parentheses. "`print`" is - as we have already mentioned - a function in version 3.x. Like any other function `print`

expects its arguments to be surrounded by parentheses. So parenthesis are an easy remedy for this error:

```
>>> print(42)
42
>>>
```

But this is not the only difference to the old print. The output behaviour has changed as well:

PRINT FUNCTION

The arguments of the print function are the following ones:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The print function can print an arbitrary number of values ("value1, value2, ..."), which are separated by commas. These values are separated by blanks. In the following example we can see two print calls. We are printing two values in both cases, i.e. a string and a float number:

```
>>> print("a = ", a)
a = 3.564
>>> print("a = \n", a)
a =
3.564
>>>
```

We can learn from the second print of the example that a blank between two values, i.e. "a = \textbackslash n" and "3.564", is always printed, even if the output is continued in the following line. This is different to Python 2, as there will be no blank printed, if a new line has been started. It's possible to redefine the separator between values by assigning an arbitrary string to the keyword parameter "sep", i.e. an empty string or a smiley:

```
>>> print("a", "b")
a b
>>> print("a", "b", sep="")
ab
>>> print(192, 168, 178, 42, sep=".")
192.168.178.42
>>> print("a", "b", sep=":-)")
a:-)b
>>>
```

A print call is ended by a newline, as we can see in the following usage:

```
>>> for i in range(4):
...     print(i)
...
0
1
2
```

```
3  
>>>
```

To change this behaviour, we can assign an arbitrary string to the keyword parameter "end". This string will be used for ending the output of the values of a print call:

```
>>> for i in range(4):  
...     print(i, end=" ")  
...  
0 1 2 3 >>>  
>>> for i in range(4):  
...     print(i, end=" :-) ")  
...  
0 :-) 1 :-) 2 :-) 3 :-) >>>
```

The output of the print function is send to the standard output stream (sys.stdout) by default. By redefining the keyword parameter "file" we can send the output into a different stream e.g. sys.stderr or a file:

```
>>> fh = open("data.txt", "w")  
>>> print("42 is the answer, but what is the question?", file=fh)  
>>> fh.close()  
>>>
```

We can see that we don't get any output in the interactive shell. The output is sent to the file "data.txt". It's also possible to redirect the output to the standard error channel this way:

```
>>> import sys  
>>> # output into sys.stderr:  
...  
>>> print("Error: 42", file=sys.stderr)  
Error: 42
```

FORMATTED OUTPUT

MANY WAYS FOR A NICER OUTPUT

In this chapter of our Python tutorial we will have a closer look at the various ways of creating nicer output in Python. We present all the different ways, but we recommend that you should use the `format` method of the string class, which you will find at end of the chapter. "string format" is by far the most flexible and Pythonic approach.

So far we have used the `print` function in two ways, when we had to print out more than two values:

- The easiest way, but not the most elegant one:

We used `print` with a comma separated list of values to print out the results, as we can see in the following example.

All the values are separated by blanks, which is the default behaviour. We can change the default value to an arbitrary string, if we assign this string to the keyword parameter "`sep`" of the `print` function:

```
>>> q = 459
>>> p = 0.098
>>> print(q, p, p * q)
459 0.098 44.982
>>> print(q, p, p * q, sep=", ")
459,0.098,44.982
>>> print(q, p, p * q, sep=" :- ")
459 :- 0.098 :- 44.982
>>>
```

- Alternatively, we can construct out of the values a new string by using the string concatenation operator:

```
>>> print(str(q) + " " + str(p) + " " + str(p * q))
459 0.098 44.982
```



>>>

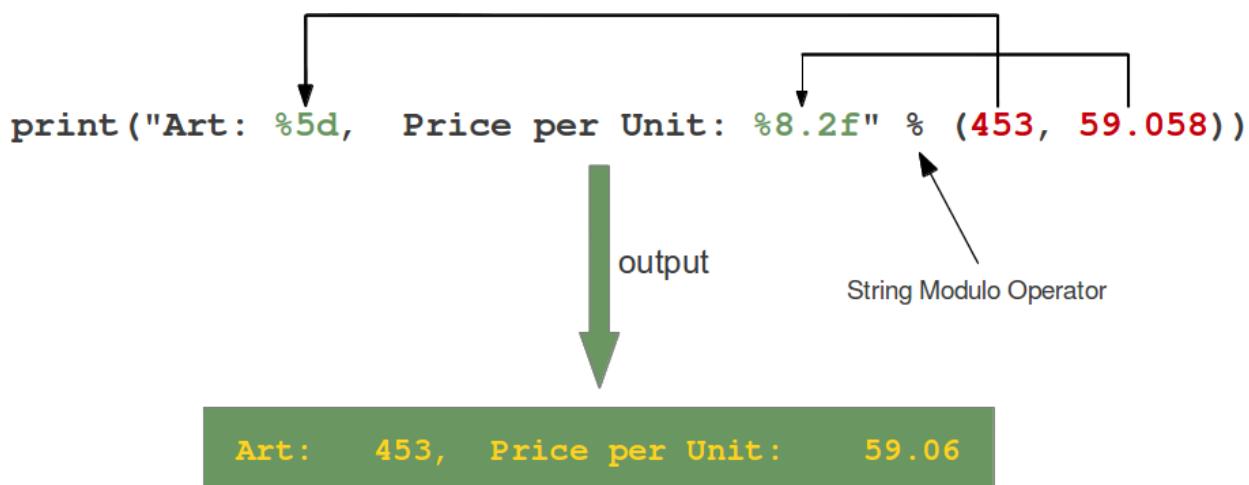
The second method is inferior to the first one in this example.

THE OLD WAY OR THE NON-EXISTING PRINTF AND SPRINTF

Is there a printf in Python? A burning question for Python newbies coming from C, Perl, Bash or other programming languages who have this statement or function. To answer that Python has a print function and no printf function is only one side of the coin or half of the truth. One can go as far as to say that this answer is not true. So there is a "printf" in Python? No, but the functionality of the "ancient" printf is contained in Python. To this purpose the modulo operator "%" is overloaded by the string class to perform string formatting. Therefore, it is often called string modulo (or sometimes even called modulus) operator, though it has not a lot in common with the actual modulo calculation on numbers. Another term for it is "string interpolation", because it interpolates various class types (like int, float and so on) into a formatted string. In many cases the string created via the string interpolation mechanism is used for outputting values in a special way. But it can also be used, for example, to create the right format to put the data into a database.

Since Python 2.6 has been introduced, the string method format should be used instead of this old-style formatting. Unfortunately, string modulo "%" is still available in Python3 and what is even worse, it is still widely used. That's why we cover it in great detail in this tutorial. You should be capable of understanding it, when you encounter it in some Python code. But it is very likely that one day this old style of formatting will be removed from the language. So you should get used to str.format().

The following diagram depicts how the string modulo operator works:



On the left side of the "string modulo operator" is the so-called format string and on the

right side is a tuple with the content, which is interpolated in the format string. The values can be literals, variables or arbitrary arithmetic expressions.

```
print("Art: %5d, Price per Unit: %8.2f" % (453, 59.058))
```

The diagram shows the code `print("Art: %5d, Price per Unit: %8.2f" % (453, 59.058))`. A green bracket underlines the entire string "Art: %5d, Price per Unit: %8.2f". Another green bracket underlines the part "% (453, 59.058)". Labels below the first bracket say "Format String" and below the second say "Tuple with values". Between the two brackets is the "% operator" with an upward arrow pointing to it.

The format string contains placeholders. There are two of those in our example: "%5d" and "%8.2f".

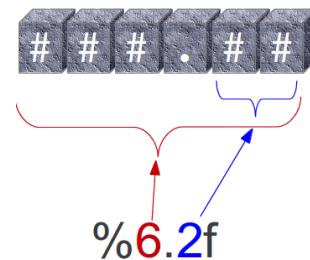
The general syntax for a format placeholder is

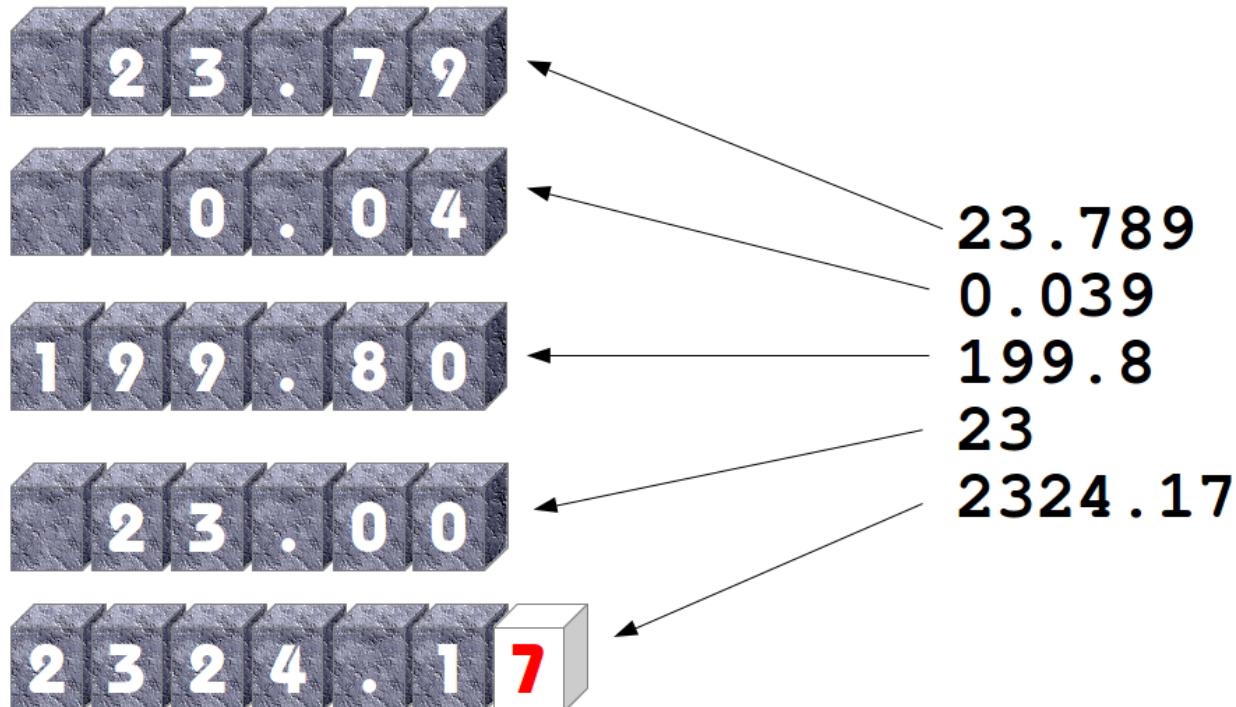
```
%[flags] [width] [.precision]type
```

Let's have a look at the placeholders in our example. The second one "%8.2f" is a format description for a float number. Like other placeholders, it is introduced with the "%" character. This is followed by the total number of digits the string should contain. This number includes the decimal point and all the digits, i.e. before and after the decimal point. Our float number 59.058 has to be formatted with 8 characters. The decimal part of the number or the precision is set to 2, i.e. the number following the "." in our placeholder. Finally, the last character "f" of our placeholder stands for "float".

If you look at the output, you will notice that the 3 decimal digits have been rounded. Furthermore, the number has been preceded in the output with 3 leading blanks.

The first placeholder "%5d" is used for the first component of our tuple, i.e. the integer 453. The number will be printed with 5 characters. As 453 consists only of 3 digits, the output is padded with 2 leading blanks. You may have expected a "%5i" instead of "%5d". Where is the difference? It's easy: There is no difference between "d" and "i" both are used for formatting integers. The advantage or beauty of a formatted output can only be seen, if more than one line is printed with the same pattern. In the following picture you can see, how it looks, if 5 float numbers are printed with the placeholder "%6.2f" are printed in subsequent lines:





Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Obsolete and equivalent to 'd', i.e. signed integer decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).

Conversion	Meaning
r	String (converts any python object using <code>repr()</code>).
s	String (converts any python object using <code>str()</code>).
%	No argument is converted, results in a "%" character in the result.

The following examples show some example cases of the conversion rules from the table above:

```
>>> print("%10.3e"% (356.08977))
      3.561e+02
>>> print("%10.3E"% (356.08977))
      3.561E+02
>>> print("%10o"% (25))
      31
>>> print("%10.3o"% (25))
      031
>>> print("%10.5o"% (25))
      00031
>>> print("%5x"% (47))
      2f
>>> print("%5.4x"% (47))
      002f
>>> print("%5.4X"% (47))
      002F
>>> print("Only one percentage sign: %% " % ())
Only one percentage sign: %
>>>
```

Flag	Meaning
#	Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively.
0	The conversion result will be zero padded for numeric values.
-	The converted value is left adjusted
	If no sign (minus sign e.g.) is going to be written, a blank space is inserted before the value.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

Examples:

```
>>> print("%#5X"% (47))
0X2F
>>> print("%5X"% (47))
2F
>>> print("%#5.4X"% (47))
0X002F
>>> print("%#5o"% (25))
0o31
>>> print("%+d"% (42))
+42
>>> print("% d"% (42))
42
>>> print("%+2d"% (42))
+42
>>> print("% 2d"% (42))
42
>>> print("%2d"% (42))
42
```

Even though it may look so, the formatting is not part of the print function. If you have a closer look at our examples, you will see that we passed a formatted string to the print function. Or to put it in other words: If the string modulo operator is applied to a string, it returns a string. This string in turn is passed in our examples to the print function. So, we could have used the string modulo functionality of Python in a two layer approach as well, i.e. first create a formatted string, which will be assigned to a variable and this variable is passed to the print function:

```
>>> s = "Price: $ %8.2f"% (356.08977)
>>> print(s)
Price: $    356.09
>>>
```

THE PYTHONIC WAY: THE STRING METHOD "FORMAT"

The Python help function is not very helpful concerning the string format method. All it says is this:

```
|     format(*args, **kwargs) -> str
|
|         Return a formatted version of S, using substitutions from
args and kwargs.
|             The substitutions are identified by braces ('{' and '}').
```

Let's dive into this topic a little bit deeper: The `format` method was added in Python 2.6. The general form of this method looks like this:

```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

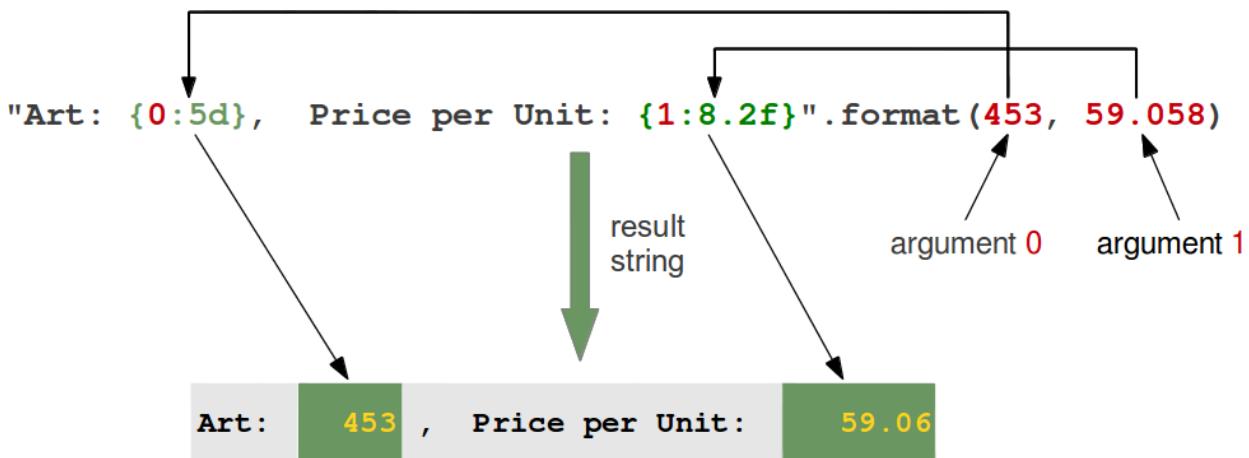
The template (or format string) is a string which contains one or more format codes (fields to be replaced) embedded in constant text. The "fields to be replaced" are surrounded by curly braces `{}`. The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments, according to the rules which we will specify soon. Anything else, which is not contained in curly braces will be literally printed, i.e. without any changes. If a brace character has to be printed, it has to be escaped by doubling it: `{}{}` and `{}{}`.

There are two kinds of arguments for the `.format()` method. The list of arguments starts with zero or more positional arguments (`p0, p1, ...`), it may be followed by zero or more keyword arguments of the form `name=value`.

A positional parameter of the `format` method can be accessed by placing the index of the parameter after the opening brace, e.g. `{0}` accesses the first parameter, `{1}` the second one and so on. The index inside of the curly braces can be followed by a colon and a format string, which is similar to the notation of the string modulo, which we had discussed in the beginning of the chapter of our tutorial, e.g. `{0:5d}`

If the positional parameters are used in the order in which they are written, the positional argument specifiers inside of the braces can be omitted, so `'{} {} {}'` corresponds to `'{0} {1} {2}'`. But they are needed, if you want to access them in different orders: `'{2} {1} {0}'`.

The following diagram with an example usage depicts how the string method "format" works for positional parameters:

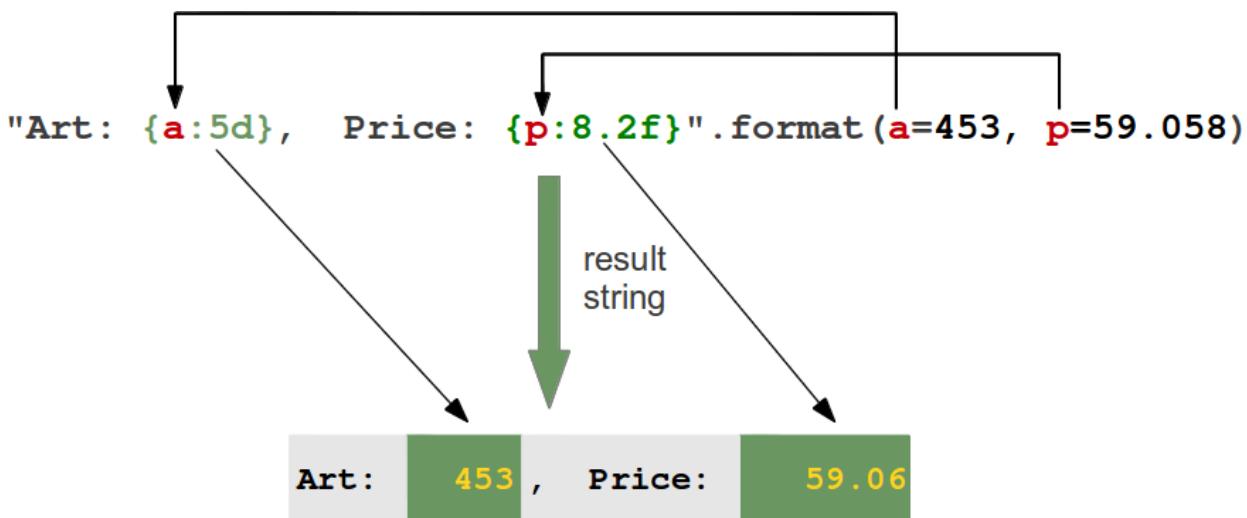


Examples of positional parameters:

```
>>> "First argument: {0}, second one: {1}".format(47,11)
'First argument: 47, second one: 11'
>>> "Second argument: {1}, first one: {0}".format(47,11)
'Second argument: 11, first one: 47'
>>> "Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
'Second argument: 11, first one: 47.42'
>>> "First argument: {}, second one: {}".format(47,11)
'First argument: 47, second one: 11'
>>> # arguments can be used more than once:
...
>>> "various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
'very various precisions: 1.41 or 1.415'
>>>
```

In the following example we demonstrate how keyword parameters can be used with the format method:

```
>>> "Art: {a:5d}, Price: {p:8.2f}".format(a=453, p=59.058)
'Art: 453, Price: 59.06'
>>>
```



It's possible to left or right justify data with the format method. To this end, we can precede the formatting with a "<" (left justify) or ">" (right justify). We demonstrate this with the following examples:

```
>>> "{0:<20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
'Spam & Eggs:       6.99'
>>> "{0:>20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
'        Spam & Eggs: 6.99'
```

Option	Meaning
'<'	The field will be left-aligned within the available space. This is usually the default for strings.
'>'	The field will be right-aligned within the available space. This is the default for numbers.
'0'	If the width field is preceded by a zero ('0') character, sign-aware zero-padding for numeric types will be enabled. <pre>>>> x = 378 >>> print("The value is {:06d}".format(x)) The value is 000378 >>> x = -378 >>> print("The value is {:06d}".format(x)) The value is -00378</pre>
', '	This option signals the use of a comma for a thousands separator. <pre>>>> print("The value is {:.}.".format(x)) The value is 78,962,324,245 >>> print("The value is {0:6,d}.".format(x)) The value is 5,897,653,423 >>> x = 5897653423.89676 >>> print("The value is {0:12,.3f}.".format(x)) The value is 5,897,653,423.897</pre>
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types.
'^'	Forces the field to be centered within the available space.

Unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

Additionally, we can modify the formatting with the *sign* option, which is only valid for number types:

Option	Meaning

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers, which is the default behavior.
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

USING DICTIONARIES IN "FORMAT"

We have seen in the previous chapters that we have two ways to access the values to be formatted:

- Using the position or the index:

```
>>> print("The capital of {0:s} is
{1:s}".format("Ontario","Toronto"))
The capital of Ontario is Toronto
>>>
```

Just to mention it once more: We could have used empty curly braces in the previous example!

- Using keyword parameters:

```
>>> print("The capital of {province} is
{capital}".format(province="Ontario",capital="Toronto"))
The capital of Ontario is Toronto
>>>
```

The second case can be expressed with a dictionary as well, as we can see in the following code:

```
>>> data = dict(province="Ontario",capital="Toronto")
>>> data
{'province': 'Ontario', 'capital': 'Toronto'}
>>> print("The capital of {province} is {capital}".format(**data))
The capital of Ontario is Toronto
```

The double "*" in front of data turns data automatically into the form 'province="Ontario",capital="Toronto"'. Let's look at the following Python program:

```
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
```

```

    "France" : "Paris",
    "England" : "London",
    "UK" : "London",
    "Switzerland" : "Bern",
    "Austria" : "Vienna",
    "Netherlands" : "Amsterdam" }

print("Countries and their capitals:")
for c in capital_country:
    print("{country}: {capital}".format(country=c,
capital=capital_country[c]))

```

If we start this program, we get the following output:

```

$ python3 country_capitals.py
Countries and their capitals:
United States: Washington
Canada: Ottawa
Austria: Vienna
Netherlands: Amsterdam
Germany: Berlin
UK: London
Switzerland: Bern
England: London
US: Washington
France: Paris

```

We can rewrite the previous example by using the dictionary directly. The output will be the same:

```

capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
                   "Austria" : "Vienna",
                   "Netherlands" : "Amsterdam" }

print("Countries and their capitals:")
for c in capital_country:
    format_string = c + ": {" + c + "}"
    print(format_string.format(**capital_country))

```

USING LOCAL VARIABLE NAMES IN "FORMAT"

"locals" is a function, which returns a dictionary with the current scope's local variables, i.e- the local variable names are the keys of this dictionary and the corresponding values are the values of these variables:

```
>>> a = 42
>>> b = 47
>>> def f(): return 42
...
>>> locals()
{'a': 42, 'b': 47, 'f': <function f at 0xb718ca6c>, '__builtins__': <module 'builtins' (built-in)>, '__package__': None, '__name__': '__main__', '__doc__': None}
>>>
```

The dictionary returned by `locals()` can be used as a parameter of the string format method. This way we can use all the local variable names inside of a format string.

Continuing with the previous example, we can create the following output, in which we use the local variables `a`, `b` and `f`:

```
>>> print("a={a}, b={b} and f={f}".format(**locals()))
a=42, b=47 and f=<function f at 0xb718ca6c>
```

OTHER STRING METHODS FOR FORMATTING

The string class contains further methods, which can be used for formatting purposes as well: `ljust`, `rjust`, `center` and `zfill`

Let `S` be a string, the 4 methods are defined like this:

- `center(...)`:

```
S.center(width[, fillchar]) -> str
```

Return `S` centred in a string of length `width`. Padding is done using the specified fill character. The default value is a space.

Examples:

```
>>> s = "Python"
>>> s.center(10)
' Python '
>>> s.center(10, "*")
'*Python*'
```

- `ljust(...)`:

```
S.ljust(width[, fillchar]) -> str
```

Return `S` left-justified in a string of length "width". Padding is done using the specified fill character. If none is given, a space will be used as default.

Examples:

```
>>> s = "Training"
>>> s.ljust(12)
'Training      '
>>> s.ljust(12, ":")
'Training::::'
>>>
```

- `rjust(...)`:

```
S.rjust(width[, fillchar]) -> str
```

Return S right-justified in a string of length width. Padding is done using the specified fill character. The default value is again a space.

Examples:

```
>>> s = "Programming"
>>> s.rjust(15)
'      Programming'
>>> s.rjust(15, "~")
'~~~~~Programming'
>>>
```

- `zfill(...)`:

```
S.zfill(width) -> str
```

Pad a string S with zeros on the left, to fill a field of the specified width. The string S is never truncated. This method can be easily emulated with `rjust`.

Examples:

```
>>> account_number = "43447879"
>>> account_number.zfill(12)
'000043447879'
>>> # can be emulated with rjust:
...
>>> account_number.rjust(12, "0")
'000043447879'
>>>
```

FORMATTED STRING LITERALS

Python 3.6 introduces formatted string literals. They are prefixed with an 'f'. The formatting syntax is similar to the format strings accepted by `str.format()`. Like the format string of

format method, they contain replacement fields formed with curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol. It's easiest to understand by looking at the following examples:

```
>>> price = 11.23
>>> f"Price in Euro: {price}"
'Price in Euro: 11.23'
>>> f"Price in Swiss Franks: {price * 1.086}"
'Price in Swiss Franks: 12.195780000000001'
>>> f"Price in Swiss Franks: {price * 1.086:5.2f}"
'Price in Swiss Franks: 12.20'
>>> for article in ["bread", "butter", "tea"]:
...     print(f"{article:>10}:")

...
    bread:
    butter:
        tea:
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

FUNCTIONS

SYNTAX

The concept of a function is one of the most important ones in mathematics. A common usage of functions in computer languages is to implement mathematical functions. Such a function is computing one or more results, which are entirely determined by the parameters passed to it.

In the most general sense, a function is a structuring element in programming languages to group a set of statements so they can be utilized more than once in a program. The only way to accomplish this without functions would be to reuse code by copying it and adapt it to its different context. Using functions usually enhances the comprehensibility and quality of the program. It also lowers the cost for development and maintenance of the software.

Functions are known under various names in programming languages, e.g. as subroutines, routines, procedures, methods, or subprograms.

A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called.

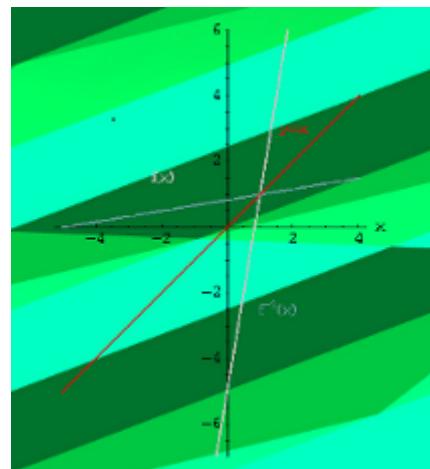
Parameter can be mandatory or optional. The optional parameters (zero or more) must follow the mandatory parameters.

Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value None is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value "None" will be returned.

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 this script looks like this:

```

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

```

OPTIONAL PARAMETERS

Functions can have optional parameters, also called default parameters. Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used. We will demonstrate the operating principle of default parameters with an example. The following little script, which isn't very useful, greets a person. If no name is given, it will greet everybody:

```

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

```

The output looks like this:

```

Hello Peter!
Hello everybody!

```

DOCSTRING

The first statement in the body of a function is usually a string, which can be accessed with function `__name__.__doc__`

This statement is called **Docstring**.

Example:

```

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

```

The output:

The docstring of the function Hello: Greets a person

KEYWORD PARAMETERS

Using keyword parameters is an alternative way to make function calls. The definition of the function doesn't change.

An example:

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

print(sumsub(12, 4))
print(sumsub(42, 15, d=10))
```

Keyword parameters can only be those, which are not used as positional arguments. We can see the benefit in the example. If we hadn't keyword parameters, the second call to function would have needed all four arguments, even though the c needs just the default value:

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

RETURN VALUES

In our previous examples, we used a return statement in the function sumsub but not in Hello. So, we can see that it is not mandatory to have a return statement. But what will be returned, if we don't explicitly give a return statement. Let's see:

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

res = no_return(4, 5)
print(res)
```

If we start this little script, None will be printed, i.e. the special value None will be returned by a return-less function. None will also be returned, if we have just a return in a function without an expression:

```
def empty_return(x, y):
    c = x + y
    return

res = empty_return(4, 5)
print(res)
```

Otherwise the value of the expression following return will be returned. In the next example 9 will be printed:

```
def return_sum(x,y):
    c = x + y
    return c

res = return_sum(4,5)
print(res)
```

RETURNING MULTIPLE VALUES

A function can return exactly one value, or we should better say one object. An object can be a numerical value, like an integer or a float. But it can also be e.g. a list or a dictionary. So, if we have to return, for example, 3 integer values, we can return a list or a tuple with these three integer values. This means that we can indirectly return multiple values. The following example, which is calculating the Fibonacci boundary for a positive number, returns a 2-tuple. The first element is the Largest Fibonacci Number smaller than x and the second component is the Smallest Fibonacci Number larger than x. The return value is immediately stored via unpacking into the variables lub and sup:

```
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))
```

LOCAL AND GLOBAL VARIABLES IN FUNCTIONS

Variable names are by default local to the function, in which they get defined.

```
def f():
    print(s)
```



```
s = "Python"
f()
```

```
def f():
    s =
"Perl"
    print(s)

s = "Python"
f()
print(s)
```

```
def f():
    print(s)
    s =
"Perl"
    print(s)

s = "Python"
f()
print(s)
```

```
def f():
    global s
    print(s)
    s =
"dog"
    print(s)
s = "cat"
f()
print(s)
```

Output:
Perl
Python

If we execute the previous script, we get the error message:

`UnboundLocalError: local variable 's'
referenced before assignment`

The variable `s` is ambiguous in `f()`, i.e. in the first `print` in `f()` the global `s` could be used with the value "Python". After this we define a local variable `s` with the assignment `s = "Perl"`

We made the variable `s` global inside of the script on the left side. Therefore anything we do to `s` inside of the function body of `f` is done to the global variable `s` outside of `f`.

Output:
cat
dog
dog

ARBITRARY NUMBER OF PARAMETERS

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple reference. This asterisk shouldn't be mistaken with the C syntax, where this notation is connected with pointers.

Example:

```
def arithmetic_mean(first, *values):
    """ This function calculates the arithmetic mean of a non-empty
    arbitrary number of numerical values """

```

```

    return (first + sum(values)) / (1 + len(values))

print(arithmetic_mean(45, 32, 89, 78))
print(arithmetic_mean(8989.8, 78787.78, 3453, 78778.73))
print(arithmetic_mean(45, 32))
print(arithmetic_mean(45))

```

Results:

```

61.0
42502.3275
38.5
45.0

```

This is great, but we have still have one problem. You may have a list of numerical values. Like, for example,

```
x = [3, 5, 9]
```

You cannot call it with

```
arithmetic_mean(x)
```

because "arithmetic_mean" can't cope with a list. Calling it with

```
arithmetic_mean(x[0], x[1], x[2])
```

is cumbersome and above all impossible inside of a program, because list can be of arbitrary length.

The solution is easy. We add a star in front of the x, when we call the function.

```
arithmetic_mean(*x)
```

This will "unpack" or singularize the list.

A practical example:

We have a list

```

my_list = [('a', 232),
            ('b', 343),
            ('c', 543),
            ('d', 23)]

```

We want to turn this list into the following list:

```
[('a', 'b', 'c', 'd'),
 (232, 343, 543, 23)]
```

This can be done by using the *-operator and the zip function in the following way:

```
list(zip(*my_list))
```

ARBITRARY NUMBER OF KEYWORD PARAMETERS

In the previous chapter we demonstrated how to pass an arbitrary number of positional parameters to a function. It is also possible to pass an arbitrary number of keyword parameters to a function. To this purpose, we have to use the double asterisk "##"

```
>>> def f(**kwargs):
...     print(kwargs)
...
>>> f()
{}
>>> f(de="German",en="English",fr="French")
{'fr': 'French', 'de': 'German', 'en': 'English'}
>>>
```

One use case is the following:

```
>>> def f(a,b,x,y):
...     print(a,b,x,y)
...
>>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
>>> f(**d)
('append', 'block', 'extract', 'yes')
```

RECURSIVE FUNCTIONS

DEFINITION

Recursion has something to do with infinity. I know recursion has something to do with infinity. I think I know recursion has something to do with infinity. He is sure I think I know recursion has something to do with infinity. We doubt he is sure I think I know ...

We think that you think that we convinced you now that we can go on forever with this example of a recursion from natural language. Recursion is not only a fundamental feature of natural language, but of the human cognitive capacity. Our way of thinking is based on a recursive thinking processes. Even with a very simple grammar, like "An English sentence contains a subject and a predicate, and a predicate contains a verb, an object and a complement", we can demonstrate the infinite possibilities of the natural language. The cognitive scientist and linguist Stephen Pinker phrases it like this: "With a few thousand nouns that can fill the subject slot and a few thousand verbs that can fill the predicate slot, one already has several million ways to open a sentence. The possible combinations quickly multiply out to unimaginably large numbers. Indeed, the repertoire of sentences is theoretically infinite, because the rules of language use a trick called recursion. A recursive rule allows a phrase to contain an example of itself, as in *She thinks that he thinks that they think that he knows* and so on, ad infinitum. And if the number of sentences is infinite, the number of possible thoughts and intentions is infinite too, because virtually every sentence expresses a different thought or intention."¹



We have to stop our short excursion to the use of recursion in natural language to come back to recursion in computer science and programs and finally to recursion in the programming language Python.

The adjective "recursive" originates from the Latin verb "recurrere", which means "to run back". And this is what a recursive definition or a recursive function does: It is "running back" or returning to itself. Most people who have done some mathematics, computer science or read a book about programming will have encountered the factorial, which is defined in mathematical terms as

$$n! = n * (n-1)!, \text{ if } n > 1 \text{ and } 0! = 1$$

It's used so often as an example for recursion because of its simplicity and clarity. We will

come back to it in the following.

DEFINITION OF RECURSION

Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition:

A recursive function has to fulfil an important condition to be used in a program: it has to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can end up in an infinite loop, if the base case is not met in the calls.

Example:

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

Replacing the calculated values gives us the following expression

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

Generally we can say: Recursion in computer science is a method where the solution to a problem is based on solving smaller instances of the same problem.

RECURSIVE FUNCTIONS IN PYTHON

Now we come to implement the factorial in Python. It's as easy and elegant as the mathematical definition.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

We can track how the function works by adding two print() functions to the previous function definition:

```

def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(", n-1,
"):", res)
        return res

print(factorial(5))

```

This Python script outputs the following results:

```

factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for 2 * factorial( 1 ): 2
intermediate result for 3 * factorial( 2 ): 6
intermediate result for 4 * factorial( 3 ): 24
intermediate result for 5 * factorial( 4 ): 120
120

```

Let's have a look at an iterative version of the factorial function.

```

def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result

```

It is common practice to extend the factorial function for 0 as an argument. It makes sense to define $0!$ to be 1, because there is exactly one permutation of zero objects, i.e. if nothing is to permute, "everything" is left in place. Another reason is that the number of ways to choose n elements among a set of n is calculated as $n!$ divided by the product of $n!$ and $0!$.

All we have to do to implement this is to change the condition of the if statement:

```

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

```

THE PITFALLS OF RECURSION

This subchapter of our tutorial on recursion deals with the Fibonacci numbers. What do have sunflowers, the Golden ratio, fir tree cones, The Da Vinci Code, the song "Lateralus" by Tool, and the graphic on the right side in common. Right, the Fibonacci numbers.

The Fibonacci numbers are the numbers of the following sequence of integer values:

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$

The Fibonacci numbers are defined by:

$$F_n = F_{n-1} + F_{n-2}$$

with $F_0 = 0$ and $F_1 = 1$

The Fibonacci sequence is named after the mathematician Leonardo of Pisa, who is better known as Fibonacci. In his book "Liber Abaci" (published 1202) he introduced the sequence as an exercise dealing with bunnies. His sequence of the Fibonacci numbers begins with $F_1 = 1$, while in modern mathematics the sequence starts with $F_0 = 0$. But this has no effect on the other members of the sequence.

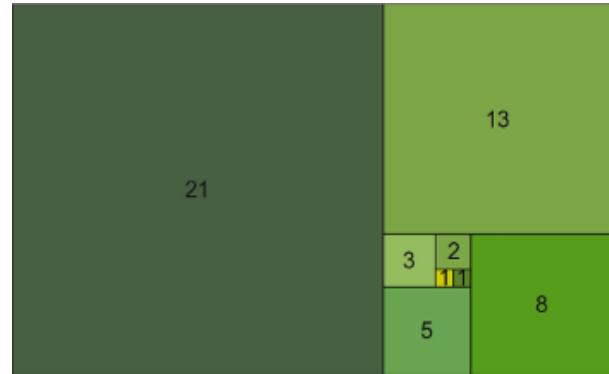
The Fibonacci numbers are the result of an artificial rabbit population, satisfying the following conditions:

- a newly born pair of rabbits, one male, one female, build the initial population
- these rabbits are able to mate at the age of one month so that at the end of its second month a female can bring forth another pair of rabbits
- these rabbits are immortal
- a mating pair always produces one new pair (one male, one female) every month from the second month onwards

The Fibonacci numbers are the numbers of rabbit pairs after n months, i.e. after 10 months we will have F_{10} rabbits.

The Fibonacci numbers are easy to write as a Python function. It's more or less a one to one mapping from the mathematical definition:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



An iterative solution is also easy to write, though the recursive solution looks more like the definition:

```
def fibi(n):
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        old, new = new, old + new
    return new
```

If you check the functions `fib()` and `fibi()`, you will find out that the iterative version `fibi()` is a lot faster than the recursive version `fib()`. To get an idea of how much this "a lot faster" can be, we have written a script, which uses the `timeit` module, to measure the calls. To do this, we save the function definitions for `fib` and `fibi` in a file `fibonacci.py`, which we can import in the program (`fibonacci_runit.py`) below:

```
from timeit import Timer

t1 = Timer("fib(10)","from fibonacci import fib")

for i in range(1,41):
    s = "fib(" + str(i) + ")"
    t1 = Timer(s,"from fibonacci import fib")
    time1 = t1.timeit(3)
    s = "fibi(" + str(i) + ")"
    t2 = Timer(s,"from fibonacci import fibi")
    time2 = t2.timeit(3)
    print("n=%2d, fib: %8.6f, fibi: %7.6f, percent: %10.2f" %
(i, time1, time2, time1/time2))
```

`time1` is the time in seconds it takes for 3 calls to `fib(n)` and `time2` respectively the time for `fibi()`. If we look at the results, we can see that calling `fib(20)` three times needs about 14 milliseconds. `fibi(20)` needs just 0.011 milliseconds for 3 calls. So `fibi(20)` is about 1300 times faster than `fib(20)`.

`fib(40)` needs already 215 seconds for three calls, while `fibi(40)` can do it in 0.016 milliseconds. `fibi(40)` is more than 13 millions times faster than `fib(40)`.

n= 1, fib:	0.000004,	fibi:	0.000005,	percent:	0.81
n= 2, fib:	0.000005,	fibi:	0.000005,	percent:	1.00
n= 3, fib:	0.000006,	fibi:	0.000006,	percent:	1.00
n= 4, fib:	0.000008,	fibi:	0.000005,	percent:	1.62
n= 5, fib:	0.000013,	fibi:	0.000006,	percent:	2.20
n= 6, fib:	0.000020,	fibi:	0.000006,	percent:	3.36
n= 7, fib:	0.000030,	fibi:	0.000006,	percent:	5.04
n= 8, fib:	0.000047,	fibi:	0.000008,	percent:	5.79
n= 9, fib:	0.000075,	fibi:	0.000007,	percent:	10.50
n=10, fib:	0.000118,	fibi:	0.000007,	percent:	16.50
n=11, fib:	0.000198,	fibi:	0.000007,	percent:	27.70
n=12, fib:	0.000287,	fibi:	0.000007,	percent:	41.52
n=13, fib:	0.000480,	fibi:	0.000007,	percent:	69.45

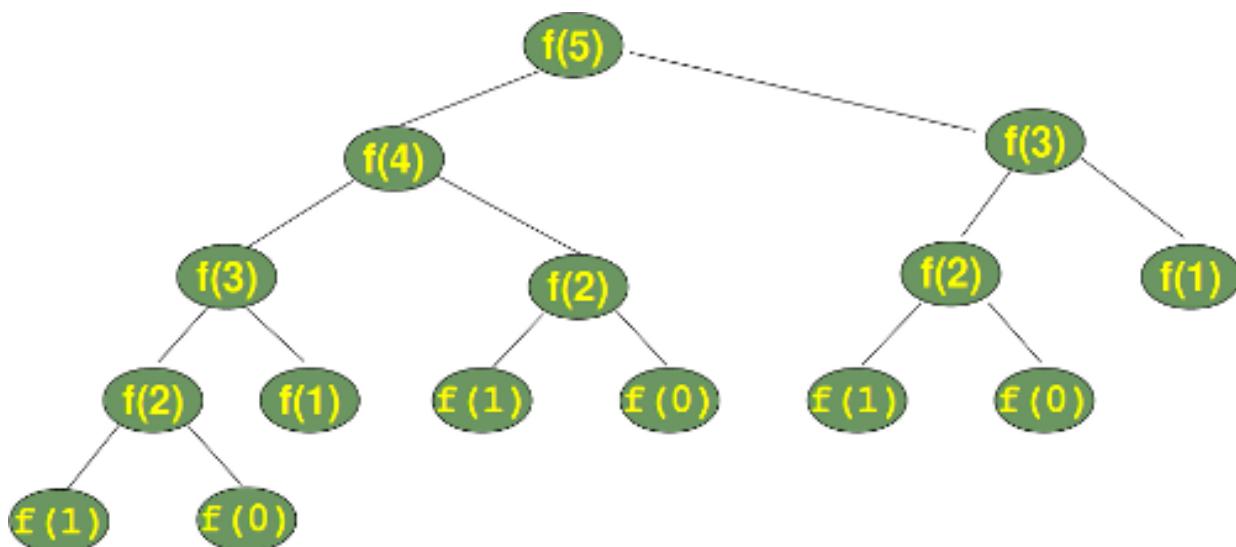
```

n=14, fib: 0.000780, fibi: 0.000007, percent: 112.83
n=15, fib: 0.001279, fibi: 0.000008, percent: 162.55
n=16, fib: 0.002059, fibi: 0.000009, percent: 233.41
n=17, fib: 0.003439, fibi: 0.000011, percent: 313.59
n=18, fib: 0.005794, fibi: 0.000012, percent: 486.04
n=19, fib: 0.009219, fibi: 0.000011, percent: 840.59
n=20, fib: 0.014366, fibi: 0.000011, percent: 1309.89
n=21, fib: 0.023137, fibi: 0.000013, percent: 1764.42
n=22, fib: 0.036963, fibi: 0.000013, percent: 2818.80
n=23, fib: 0.060626, fibi: 0.000012, percent: 4985.96
n=24, fib: 0.097643, fibi: 0.000013, percent: 7584.17
n=25, fib: 0.157224, fibi: 0.000013, percent: 11989.91
n=26, fib: 0.253764, fibi: 0.000013, percent: 19352.05
n=27, fib: 0.411353, fibi: 0.000012, percent: 34506.80
n=28, fib: 0.673918, fibi: 0.000014, percent: 47908.76
n=29, fib: 1.086484, fibi: 0.000015, percent: 72334.03
n=30, fib: 1.742688, fibi: 0.000014, percent: 123887.51
n=31, fib: 2.861763, fibi: 0.000014, percent: 203442.44
n=32, fib: 4.648224, fibi: 0.000015, percent: 309461.33
n=33, fib: 7.339578, fibi: 0.000014, percent: 521769.86
n=34, fib: 11.980462, fibi: 0.000014, percent: 851689.83
n=35, fib: 19.426206, fibi: 0.000016, percent: 1216110.64
n=36, fib: 30.840097, fibi: 0.000015, percent: 2053218.13
n=37, fib: 50.519086, fibi: 0.000016, percent: 3116064.78
n=38, fib: 81.822418, fibi: 0.000015, percent: 5447430.08
n=39, fib: 132.030006, fibi: 0.000018, percent: 7383653.09
n=40, fib: 215.091484, fibi: 0.000016, percent: 13465060.78

```

What's wrong with our recursive implementation?

Let's have a look at the calculation tree, i.e. the order in which the functions are called. `fib()` is substituted by `f()`.



We can see that the subtree `f(2)` appears 3 times and the subtree for the calculation of `f(3)` two times. If you imagine extending this tree for `f(6)`, you will understand that `f(4)` will be called two times, `f(3)` three times and so on. This means, our recursion doesn't remember

previously calculated values.

We can implement a "memory" for our recursive version by using a dictionary to save the previously calculated values.

```
memo = {0:0, 1:1}
def fibm(n):
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```

We time it again to compare it with fibi():

```
from timeit import Timer
from fibonacci import fib

t1 = Timer("fib(10)","from fibonacci import fib")

for i in range(1,41):
    s = "fibm(" + str(i) + ")"
    t1 = Timer(s,"from fibonacci import fibm")
    time1 = t1.timeit(3)
    s = "fibi(" + str(i) + ")"
    t2 = Timer(s,"from fibonacci import fibi")
    time2 = t2.timeit(3)
    print("n=%2d, fib: %8.6f, fibi: %7.6f, percent: %10.2f" %
(i, time1, time2, time1/time2))
```

We can see that it is even faster than the iterative version. Of course, the larger the arguments the greater the benefit of our memoization:

n= 1, fib:	0.000011, fibi:	0.000015, percent:	0.73
n= 2, fib:	0.000011, fibi:	0.000013, percent:	0.85
n= 3, fib:	0.000012, fibi:	0.000014, percent:	0.86
n= 4, fib:	0.000012, fibi:	0.000015, percent:	0.79
n= 5, fib:	0.000012, fibi:	0.000016, percent:	0.75
n= 6, fib:	0.000011, fibi:	0.000017, percent:	0.65
n= 7, fib:	0.000012, fibi:	0.000017, percent:	0.72
n= 8, fib:	0.000011, fibi:	0.000018, percent:	0.61
n= 9, fib:	0.000011, fibi:	0.000018, percent:	0.61
n=10, fib:	0.000010, fibi:	0.000020, percent:	0.50
n=11, fib:	0.000011, fibi:	0.000020, percent:	0.55
n=12, fib:	0.000004, fibi:	0.000007, percent:	0.59
n=13, fib:	0.000004, fibi:	0.000007, percent:	0.57
n=14, fib:	0.000004, fibi:	0.000008, percent:	0.52
n=15, fib:	0.000004, fibi:	0.000008, percent:	0.50
n=16, fib:	0.000003, fibi:	0.000008, percent:	0.39
n=17, fib:	0.000004, fibi:	0.000009, percent:	0.45
n=18, fib:	0.000004, fibi:	0.000009, percent:	0.45
n=19, fib:	0.000004, fibi:	0.000009, percent:	0.45
n=20, fib:	0.000003, fibi:	0.000010, percent:	0.29
n=21, fib:	0.000004, fibi:	0.000009, percent:	0.45

```

n=22, fib: 0.000004, fibi: 0.000010, percent: 0.40
n=23, fib: 0.000004, fibi: 0.000010, percent: 0.40
n=24, fib: 0.000004, fibi: 0.000011, percent: 0.35
n=25, fib: 0.000004, fibi: 0.000012, percent: 0.33
n=26, fib: 0.000004, fibi: 0.000011, percent: 0.34
n=27, fib: 0.000004, fibi: 0.000011, percent: 0.35
n=28, fib: 0.000004, fibi: 0.000012, percent: 0.32
n=29, fib: 0.000004, fibi: 0.000012, percent: 0.33
n=30, fib: 0.000004, fibi: 0.000013, percent: 0.31
n=31, fib: 0.000004, fibi: 0.000012, percent: 0.34
n=32, fib: 0.000004, fibi: 0.000012, percent: 0.33
n=33, fib: 0.000004, fibi: 0.000013, percent: 0.30
n=34, fib: 0.000004, fibi: 0.000012, percent: 0.34
n=35, fib: 0.000004, fibi: 0.000013, percent: 0.31
n=36, fib: 0.000004, fibi: 0.000013, percent: 0.31
n=37, fib: 0.000004, fibi: 0.000014, percent: 0.29
n=38, fib: 0.000004, fibi: 0.000014, percent: 0.29
n=39, fib: 0.000004, fibi: 0.000013, percent: 0.31
n=40, fib: 0.000004, fibi: 0.000014, percent: 0.29

```

We can also define a recursive algorithm for our Fibonacci function by using a class with callable instances, i.e. by using the special method `__call__`. This way, we will be able to hide the dictionary in an elegant way. We used a general approach which allows us to define also functions similar to Fibonacci, like the Lucas function:

```

class Fibonacci:

    def __init__(self, i1=0, i2=1):
        self.memo = {0:i1, 1:i2}

    def __call__(self, n):
        if n not in self.memo:
            self.memo[n] = self.__call__(n-1) + self.__call__(n-2)
        return self.memo[n]

fib = Fibonacci()
lucas = Fibonacci(2, 1)

for i in range(1, 16):
    print(i, fib(i), lucas(i))

```

The program returns the following output:

```

1 1 1
2 1 3
3 2 4
4 3 7
5 5 11
6 8 18
7 13 29
8 21 47
9 34 76
10 55 123
11 89 199

```

```

12 144 322
13 233 521
14 377 843
15 610 1364

```

The Lucas numbers or Lucas series are an integer sequence named after the mathematician François Édouard Anatole Lucas (1842–91), who studied both that sequence and the closely related Fibonacci numbers. The Lucas numbers have the same creation rule than the Fibonacci number, i.e. the sum of the two previous numbers, but the values for 0 and 1 are different.

MORE ABOUT RECURSION IN PYTHON

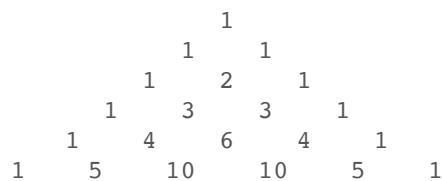
If you want to learn more on recursion, we suggest that you try to solve the following exercises. Please do not peer at the solutions, before you haven't given your best. If you have thought about a task for a while and you are still not capable of solving the exercise, you may consult our sample solutions.



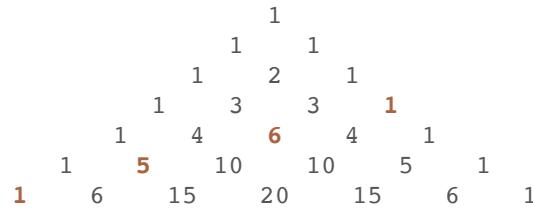
In our section "Advanced Topics" of our tutorial we have a comprehensive treatment of the game or puzzle "[Towers of Hanoi](#)". Of course, we solve it with a function using a recursive function. The "Hanoi problem" is special, because a recursive solution almost forces itself on the programmer, while the iterative solution of the game is hard to find and to grasp.

EXERCISES

1. Think of a recursive version of the function $f(n) = 3 * n$, i.e. the multiples of 3
2. Write a recursive Python function that returns the sum of the first n integers.
(Hint: The function will be similar to the factorial function!)
3. Write a function which implements the Pascal's triangle:



4. The Fibonacci numbers are hidden inside of Pascal's triangle. If you sum up the coloured numbers of the following triangle, you will get the 7th Fibonacci number:



Write a recursive program to calculate the Fibonacci numbers, using Pascal's triangle.

5. Implement a recursive function in Python for the sieve of Eratosthenes.

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a specified integer. It was created by the ancient Greek mathematician Eratosthenes. The algorithm to find all the prime numbers less than or equal to a given integer n :

1. Create a list of integers from two to n : 2, 3, 4, ..., n
2. Start with a counter i set to 2, i.e. the first prime number
3. Starting from $i+i$, count up by i and remove those numbers from the list, i.e. $2*i$, $3*i$, $4*i$, aso..
4. Find the first number of the list following i . This is the next prime number.
5. Set i to the number found in the previous step
6. Repeat steps 3 and 4 until i is greater than n . (As an improvement: It's enough to go to the square root of n)
7. All the numbers, which are still in the list, are prime numbers

You can easily see that we would be inefficient, if we strictly used this algorithm, e.g. we will try to remove the multiples of 4, although they have been already removed by the multiples of 2. So it's enough to produce the multiples of all the prime numbers up to the square root of n . We can recursively create these sets.

6. Write a recursive function `find_index()`, which returns the index of a number in the Fibonacci sequence, if the number is an element of this sequence and returns -1 if the number is not contained in it, i.e.

```
fib(find_index(n)) == n
```

7. The sum of the squares of two consecutive Fibonacci numbers is also a Fibonacci number, e.g. 2 and 3 are elements of the Fibonacci sequence and $2^2 + 3^2 = 13$ corresponds to $\text{Fib}(7)$.

Use the previous function to find the position of the sum of the squares of two consecutive numbers in the Fibonacci sequence.

Mathematical explanation:

Let a and b be two successive Fibonacci numbers with a prior to b . The Fibonacci sequence starting with the number " a " looks like this:

0	a
1	b
2	$a + b$
3	$a + 2b$

```

4      2a + 3b
5      3a + 5b
6      5a + 8b

```

We can see that the Fibonacci numbers appear as factors for a and b. The n-th element in this sequence can be calculated with the following formula:

$$F(n) = Fib(n-1) * a + Fib(n) * b$$

From this we can conclude that for a natural number n, $n > 1$, the following holds true:

$$Fib(2n + 1) = Fib(n)^2 + Fib(n+1)^2$$

SOLUTIONS TO OUR EXERCISES

1. Solution to our first exercise on recursion:

Mathematically, we can write it like this:

$$\begin{aligned} f(1) &= 3, \\ f(n+1) &= f(n) + 3 \end{aligned}$$

A Python function can be written like this:

```

def mult3(n):
    if n == 1:
        return 3
    else:
        return mult3(n-1) + 3
Towers of Hanoi
for i in range(1,10):
    print(mult3(i))

```

2. Solution to our second exercise:

```

def sum_n(n):
    if n== 0:
        return 0
    else:
        return n + sum_n(n-1)

```

3. Solution for creating the Pascal triangle:

```

def pascal(n):
    if n == 1:
        return [1]
    else:
        line = [1]
        previous_line = pascal(n-1)
        for i in range(len(previous_line)-1):
            line.append(previous_line[i] + previous_line[i+1])
        line += [1]
    return line

```

```
print(pascal(6))
```

Alternatively, we can write a function using list comprehension:

```
def pascal(n):
    if n == 1:
        return [1]
    else:
        p_line = pascal(n-1)
        line = [p_line[i]+p_line[i+1] for i in
range(len(p_line)-1)]
        line.insert(0,1)
        line.append(1)
    return line

print(pascal(6))
```

4. Producing the Fibonacci numbers out of Pascal's triangle:

```
def fib_pascal(n,fib_pos):
    if n == 1:
        line = [1]
        fib_sum = 1 if fib_pos == 0 else 0
    else:
        line = [1]
        (previous_line, fib_sum) = fib_pascal(n-1, fib_pos+1)
        for i in range(len(previous_line)-1):
            line.append(previous_line[i] + previous_line[i+1])
        line += [1]
        if fib_pos < len(line):
            fib_sum += line[fib_pos]
    return (line, fib_sum)

def fib(n):
    return fib_pascal(n,0)[1]

# and now printing out the first ten Fibonacci numbers:
for i in range(1,10):
    print(fib(i))
```

5. The following program implements the sieve of Eratosthenes according to the rules of the task in an iterative way. It will print out the first 100 prime numbers.

```
from math import sqrt

def sieve(n):
    # returns all primes between 2 and n
    primes = list(range(2,n+1))
    max = sqrt(n)
    num = 2
    while num < max:
        i = num
        while i <= n:
            i += num
            if i in primes:
                primes.remove(i)
```

```

        for j in primes:
            if j > num:
                num = j
                break
    return primes

print(sieve(100))

```

But this chapter of our tutorial is about recursion and recursive functions, and we have demanded a recursive function to calculate the prime numbers. To understand the following solution, you may confer our chapter about [List Comprehension](#):

```

from math import sqrt

def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return []
    else:
        p = primes(int(sqrt(n)))
        no_p = [j for i in p for j in range(i*2, n + 1, i)]
        p = [x for x in range(2, n + 1) if x not in no_p]
    return p

print(primes(100))

6. memo = {0:0, 1:1}
def fib(n):
    if not n in memo:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

def find_index(*x):
    """ finds the natural number i with fib(i) = n """
    if len(x) == 1:
        # started by user
        # find index starting from 0
        return find_index(x[0],0)
    else:
        n = fib(x[1])
        m = x[0]
        if n > m:
            return -1
        elif n == m:
            return x[1]
        else:
            return find_index(m,x[1]+1)

7.
# code from the previous example with the functions fib() and
find_index()

```

```

print(" index of a |      a |      b | sum of squares | index ")
print("=====|=====|=====|=====|====")
for i in range(15):
    square = fib(i)**2 + fib(i+1)**2
    print( " %10d | %3d | %3d | %14d | %5d " % (i,
fib(i), fib(i+1), square, find_index(square)))

```

The result of the previous program looks like this:

index of a	a	b	sum of squares	index
0	0	1	1	1
1	1	1	2	3
2	1	2	5	5
3	2	3	13	7
4	3	5	34	9
5	5	8	89	11
6	8	13	233	13
7	13	21	610	15
8	21	34	1597	17
9	34	55	4181	19
10	55	89	10946	21
11	89	144	28657	23
12	144	233	75025	25
13	233	377	196418	27
14	377	610	514229	29

¹ Stephen Pinker, The Blank Slate, 2002

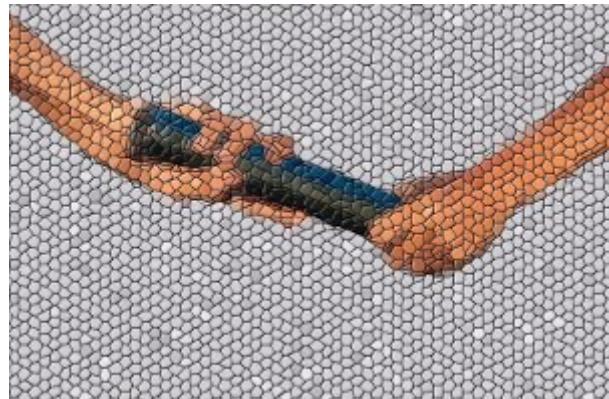
PARAMETERS AND ARGUMENTS

A function or procedure usually needs some information about the environment, in which it has been called. The interface between the environment, from which the function has been called, and the function, i.e. the function body, consists of special variables, which are called parameters. By using these parameters, it's possible to use all kind of objects from "outside" inside of a function. The syntax for how parameters are declared and the semantics for how the arguments are passed to the parameters of the function or procedure are depending on the programming language.

Very often the terms parameter and argument are used synonymously, but there is a clear difference. Parameters are inside functions or procedures, while arguments are used in procedure calls, i.e. the values passed to the function at run-time.

"CALL BY VALUE" AND "CALL BY NAME"

The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs from programming language to programming language. The most common evaluation strategies are "call by value" and "call by reference":



- **Call by Value**

The most common strategy is the call-by-value evaluation, sometimes also called pass-by-value. This strategy is used in C and C++, for example. In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will be unchanged when the function returns.

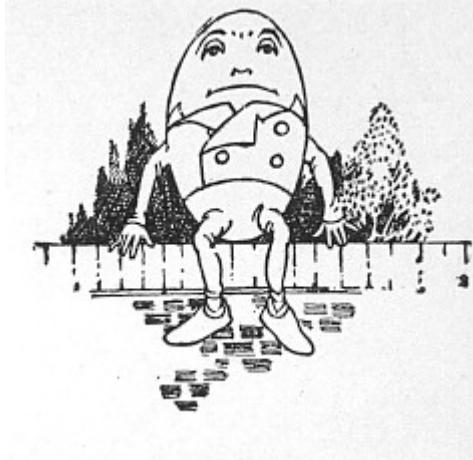
- **Call by Reference**

In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed. By using Call by Reference we save both computation time and memory space, because arguments do not need to be copied. On the other hand this harbours the disadvantage that variables can be "accidentally" changed in a function call. So special care has to be taken to "protect" the values, which shouldn't be changed.

Many programming language support call-by-reference, like C or C++, but Perl uses it as default.

In ALGOL 60 and COBOL has been known a different concept, which was called call-by-name, which isn't used anymore in modern languages.

AND WHAT ABOUT PYTHON?



There are books which call the strategy of Python call-by-value and others call it call-by-reference. You may ask yourself, what is right.

Humpty Dumpty supplies the explanation:

--- "When I use a word," Humpty Dumpty said, in a rather a scornful tone, "it means just what I choose it to mean - neither more nor less."

--- "The question is," said Alice, "whether you can make words mean so many different things."

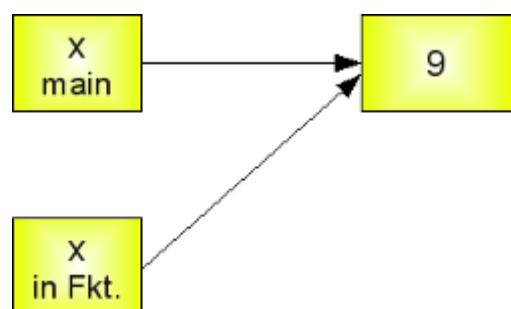
--- "The question is," said Humpty Dumpty, "which is to be master - that's all."

Lewis Carroll, Through the Looking-Glass

To come back to our initial question what evaluation strategy is used in Python: The authors who call the mechanism call-by-value and those who call it call-by-reference are stretching the definitions until they fit.

Correctly speaking, Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the

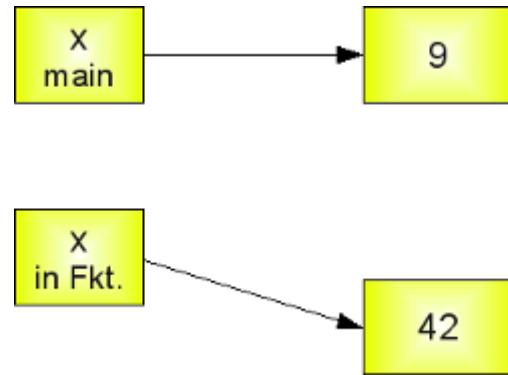


function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

First, let's have a look at the integer variables. The parameter inside of the function remains a reference to the arguments variable, as long as the parameter is not changed. As soon as a new value will be assigned to it, Python creates a separate local variable. The caller's variable will not be changed this way:

```
def ref_demo(x):
    print("x=", x, " id=", id(x))
    x=42
    print("x=", x, " id=", id(x))
```

In the example above, we used the `id()` function, which takes an object as a parameter. `id(obj)` returns the "identity" of the object "obj". This identity, the return value of the function, is an integer which is unique and constant for this object during its lifetime. Two different objects with non-overlapping lifetimes may have the same `id()` value.



If you call the function `ref_demo()` of the previous example - like we do in the green block further down - we can check with the `id()` function what happens to `x`: We can see that in the main scope, `x` has the identity 41902552. In the first print statement of the `ref_demo()` function, the `x` from the main scope is used, because we can see that we get the same identity. After we have assigned the value 42 to `x`, `x` gets a new identity 41903752, i.e. a separate memory location from the global `x`. So, when we are back in the main scope `x` has still the original value 9.

This means that Python initially behaves like call-by-reference, but as soon as we are changing the value of such a variable, i.e. as soon as we assign a new object to it, Python "switches" to call-by-value. This means that a local variable `x` will be created and the value of the global variable `x` will be copied into it.

```
>>> x = 9
>>> id(x)
9251936
>>> ref_demo(x)
x= 9  id= 9251936
x= 42  id= 9252992
>>> id(x)
9251936
>>>
```

SIDE EFFECTS

A function is said to have a side effect, if, in addition to producing a return value, it modifies the caller's environment in other ways. For example, a function might modify a global or static variable, modify one of its arguments, raise an exception, write data to a display or file and so on.

There are situations, in which these side effects are intended, i.e. they are part of the functions specification. But in other cases, they are not wanted , they are hidden side effects. In this chapter we are only interested in the side effects, which change one or more global variables, which have been passed as arguments to a function.

Let's assume, we are passing a list to a function. We expect that the function is not changing this list. First let's have a look at a function which has no side effects. As a new list is assigned to the parameter list in func1(), a new memory location is created for list and list becomes a local variable.

```
>>> def no_side_effects(cities):
...     print(cities)
...     cities = cities + ["Birmingham", "Bradford"]
...     print(cities)
...
>>> locations = ["London", "Leeds", "Glasgow", "Sheffield"]
>>> no_side_effects(locations)
['London', 'Leeds', 'Glasgow', 'Sheffield']
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham',
'Bradford']
>>> print(locations)
['London', 'Leeds', 'Glasgow', 'Sheffield']
>>>
```

This changes drastically, if we increment the list by using augmented assignment operator `+=`. To show this, we change the previous function rename it to "side_effects" in the following example:

```
>>> def side_effects(cities):
...     print(cities)
...     cities += ["Birmingham", "Bradford"]
...     print(cities)
...
>>> locations = ["London", "Leeds", "Glasgow", "Sheffield"]
>>> side_effects(locations)
['London', 'Leeds', 'Glasgow', 'Sheffield']
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham',
'Bradford']
>>> print(locations)
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham',
'Bradford']
>>>
```

We can see that Birmingham and Bradford are included in the global list locations as well, because `+=` acts as an in-place operation.

The user of this function can prevent this side effect by passing a copy to the function. A shallow copy is sufficient, because there are no nested structures in the list. To satisfy our French customers as well, we change the city names in the next example to demonstrate the effect of the slice operator in the function call:

```
>>> def side_effects(cities):
...     print(cities)
...     cities += ["Paris", "Marseille"]
...     print(cities)
...
>>> locations = ["Lyon", "Toulouse", "Nice", "Nantes", "Strasbourg"]
>>> side_effects(locations[:])
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Paris',
'Marseille']
>>> print(locations)
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
>>>
```

We can see that the global list locations has not been effected by the execution of the function.

COMMAND LINE ARGUMENTS

If you use a command line interface, i.e. a text user interface (TUI), and not a graphical user interface (GUI), command line arguments are very useful. They are arguments which are added after the function call in the same line.

It's easy to write Python scripts using command line arguments. If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces. Inside of the script these arguments are accessible through the list variable sys.argv. The name of the script is included in this list sys.argv[0]. sys.argv[1] contains the first parameter, sys.argv[2] the second and so on.

The following script (arguments.py) prints all arguments:

```
# Module sys has to be imported:
import sys

# Iteration over all arguments:
for eachArg in sys.argv:
    print(eachArg)
```

Example call to this script:

```
python argumente.py python course for beginners
```

This call creates the following output:

```
argumente.py
python
```

```
course
for
beginners
```

VARIABLE LENGTH OF PARAMETERS

We will introduce now functions, which can take an arbitrary number of arguments. Those who have some programming background in C or C++ know this from the varargs feature of these languages.

Some definitions, which are not really necessary for the following: A function with an arbitrary number of arguments is usually called a variadic function in computer science. To use another special term: A variadic function is a function of indefinite arity. The arity of a function or an operation is the number of arguments or operands that the function or operation takes. The term was derived from words like "unary", "binary", "ternary", all ending in "ary".

The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

```
>>> def varpafu(*x): print(x)
...
>>> varpafu()
()
>>> varpafu(34, "Do you like Python?", "Of course")
(34, 'Do you like Python?', 'Of course')
>>>
```

We learn from the previous example that the arguments passed to the function call of varpafu() are collected in a tuple, which can be accessed as a "normal" variable x within the body of the function. If the function is called without any arguments, the value of x is an empty tuple.

Sometimes, it's necessary to use positional parameters followed by an arbitrary number of parameters in a function definition. This is possible, but the positional parameters always have to precede the arbitrary parameters. In the following example, we have a positional parameter "city", - the main location, - which always have to be given, followed by an arbitrary number of other locations:

```
>>> def locations(city, *other_cities): print(city, other_cities)
...
>>> locations("Paris")
Paris ()
>>> locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux",
"Marseille")
Paris ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille')
>>>
```

EXERCISE

Write a function which calculates the arithmetic mean of a variable number of values.

SOLUTION

```
def arithmetic_mean(x, *l):
    """ The function calculates the arithmetic mean of a non-empty
    arbitrary number of numbers """
    sum = x
    for i in l:
        sum += i

    return sum / (1.0 + len(l))
```

You might ask yourself, why we used both a positional parameter "x" and the variable parameter "*l" in our function definition. We could have only used *l to contain all our numbers. The idea is that we wanted to enforce that we always have a non-empty list of numbers. This is necessary to prevent a division by zero error, because the average of an empty list of numbers is not defined.

In the following interactive Python session, we can learn how to use this function. We assume that the function `arithmetic_mean` is saved in a file called `statistics.py`.

```
>>> from statistics import arithmetic_mean
>>> arithmetic_mean(4, 7, 9)
6.66666666666667
>>> arithmetic_mean(4, 7, 9, 45, -3.7, 99)
26.7166666666667
```

This works fine, but there is a catch. What if somebody wants to call the function with a list, instead of a variable number of numbers, as we have shown above? We can see in the following that we raise an error, as most hopefully, you might expect:

```
>>> l = [4, 7, 9, 45, -3.7, 99]
>>> arithmetic_mean(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "statistics.py", line 8, in arithmetic_mean
    return sum / (1.0 + len(l))
TypeError: unsupported operand type(s) for /: 'list' and 'float'
```

The rescue is using another asterisk:

```
>>> arithmetic_mean(*l)
26.7166666666667
>>>
```

* IN FUNCTION CALLS

A * can appear in function calls as well, as we have just seen in the previous exercise: The semantics is in this case "inverse" to a star in a function definition. An argument will be unpacked and not packed. In other words, the elements of the list or tuple are singularized:

```
>>> def f(x,y,z):
...     print(x,y,z)
...
>>> p = (47,11,12)
>>> f(*p)
(47, 11, 12)
```

There is hardly any need to mention that this way of calling our function is more comfortable than the following one:

```
>>> f(p[0],p[1],p[2])
(47, 11, 12)
>>>
```

Additionally to being less comfortable, the previous call (`f(p[0],p[1],p[2])`) doesn't work in the general case, i.e. lists of unknown lengths. "Unknown" mean, that the length is only known at runtime and not when we are writing the script.

ARBITRARY KEYWORD PARAMETERS

There is also a mechanism for an arbitrary number of keyword parameters. To do this, we use the double asterisk "***" notation:

```
>>> def f(**args):
...     print(args)
...
>>> f()
{}
>>> f(de="German",en="English",fr="French")
{'fr': 'French', 'de': 'German', 'en': 'English'}
>>>
```

DOUBLE ASTERISK IN FUNCTION CALLS

The following example demonstrates the usage of ** in a function call:

```
>>> def f(a,b,x,y):
...     print(a,b,x,y)
...
>>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
>>> f(**d)
('append', 'block', 'extract', 'yes')
```

and now in combination with *:

```
>>> t = (47,11)
>>> d = {'x':'extract','y':'yes'}
>>> f(*t, **d)
(47, 11, 'extract', 'yes')
>>>
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

NAMESPACES AND SCOPES

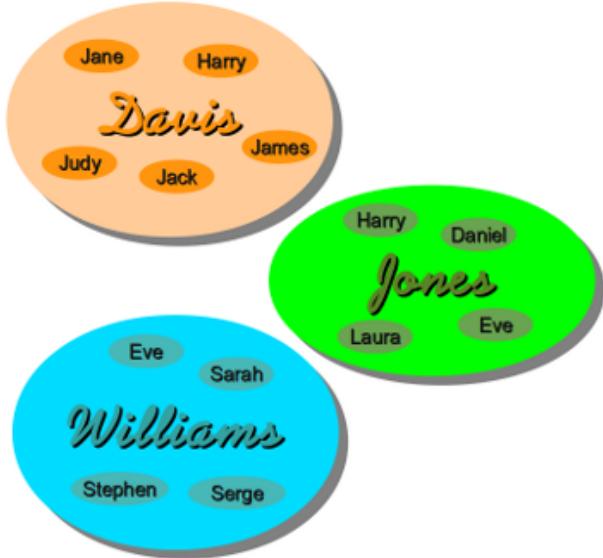
NAMESPACES

Generally speaking, a **namespace** (sometimes also called a context) is a naming system for making names unique to avoid ambiguity. Everybody knows a namespacing system from daily life, i.e. the naming of people in first name and family name (surname). Another example is a network: each network device (workstation, server, printer, ...) needs a unique name and address. Yet another example is the directory structure of filesystems. The same filename can be used in different directories, the files can be uniquely accessed via the pathnames.

Many programming languages use namespaces or contexts for identifiers. An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces. (Like the same filenames in different directories) Programming languages, which support namespaces may have different rules that determine to which namespace an identifier belongs.

Namespaces in Python are implemented as Python dictionaries, this means that they are defined by a mapping from names, i.e. the keys of the dictionary, to objects, i.e. the values. The user doesn't have to know this to write a Python program and when using namespaces. Some namespaces in Python:

- **global names** of a module
- **local names** in a function or method invocation
- **built-in names**: this namespace contains built-in functions (e.g. `abs()`, `cmp()`, ...) and built-in exception names



LIFETIME OF A NAMESPACE

Not every namespace, which may be used in a script or program is accessible (or alive) at any moment during the execution of the script. Namespaces have different lifetimes, because they are often created at different points in time. There is one namespace which is present from beginning to end: The namespace containing the built-in names is created

when the Python interpreter starts up, and is never deleted. The global namespace of a module is generated when the module is read in. Module namespaces normally last until the script ends, i.e. the interpreter quits. When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

SCOPES

A scope refers to a region of a program where a namespace can be directly accessed, i.e. without using a namespace prefix. In other words: The scope of a name is the area of a program where this name can be unambiguously used, for example, inside of a function. A name's namespace is identical to its scope. Scopes are defined statically, but they are used dynamically.

During program execution there are the following nested scopes available:

- the innermost scope is searched first and it contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope
- the next-to-last scope contains the current module's global names
- the outermost scope, which is searched last, is the namespace containing the built-in names

GLOBAL, LOCAL AND NONLOCAL VARIABLES

The way Python uses global and local variables is maverick. While in many or most other programming languages variables are treated as global if not otherwise declared, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out. Like in many other program structures, Python also imposes good programming habit by design.



So when you define variables inside a function definition, they are local to this function by default. This means that anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. This means that the function body is the scope of such a variable, i.e. the enclosing context where this name with its values is associated.

All variables have the scope of the block, where they are declared and defined in. They can only be used after the point of their declaration.

Just to make things clear: Variables don't have to be and can't be declared in the way they are declared in programming languages like Java or C. Variables in Python are implicitly declared by defining them, i.e. the first time you assign a value to a variable, this variable is declared and has automatically the data type of the object which has to be assigned to it. If you have problems understanding this, please consult our chapter about data types and variables, see links on the left side.

GLOBAL AND LOCAL VARIABLES IN FUNCTIONS

In the following example, we want to demonstrate, how global values can be used inside the body of a function:

```
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

The variable s is defined as the string "I love Paris in the summer!", before calling the function f(). The body of f() consists solely of the "print(s)" statement. As there is no local variable s, i.e. no assignment to s, the value from the global variable s will be used. So the output will be the string "I love Paris in the summer!". The question is, what will happen, if we change the value of s inside of the function f()? Will it affect the global variable as well? We test this in the following piece of code:

```
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
print(s)
```

The output looks like this:

```
I love London!
I love Paris!
```

What if we combine the first example with the second one, i.e. we first access s with a print() function, hoping to get the global value, and then assigning a new value to it? Assigning a value to it, means - as we have previously stated - creating a local variable s. So, we would have s both as a global and a local variable in the same scope, i.e. the body of the function. Python fortunately doesn't allow this ambiguity. So, it will throw an error, as we can see in the following example:

```
>>> def f():
...     print(s)
...     s = "I love London!"
...     print(s)
...
>>> s = "I love Paris!"
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 's' referenced before assignment
>>>
```

A variable can't be both local and global inside of a function. So Python decides that we want a local variable due to the assignment to s inside of f(), so the first print statement before the definition of s throws the error message above. Any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global", as can be seen in the following example:

```
def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)
```

```
s = "I am looking for a course in Paris!"
f()
print(s)
```

We have solved our problem. There is no ambiguity left. The output of this small script looks like this:

```
I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

Local variables of functions can't be accessed from outside, when the function call has finished:

```
def f():
    s = "I am globally not known"
    print(s)

f()
print(s)
```

Starting this script gives us the following output with an error message:

```
monty@python:~$ python3 ex.py
I am globally not known
Traceback (most recent call last):
  File "ex.py", line 6, in <module>
    print(s)
NameError: name 's' is not defined
monty@python:~$
```

The following example shows a wild combination of local and global variables and function parameters:

```
def foo(x, y):
    global a
    a = 42
    x, y = y, x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3, 4
foo(17, 4)
print(a, b, x, y)
```

The output looks like this:

```
42 17 4 17
42 15 3 4
```

GLOBAL VARIABLES IN NESTED FUNCTIONS

We will examine now what will happen, if we use the global keyword inside of nested functions. The following example shows a situation where a variable x is used in various scopes:

```
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```

This program returns the following results:

```
Before calling g: 42
Calling g now:
After calling g: 42
x in main: 43
```

We can see that the global statement inside of the nested function g does not affect the variable x of the function f, i.e. it keeps its value 42. We can also deduce from this example that after calling f() a variable x exists in the module namespace and has the value 43. This means that the global keyword in nested functions does not affect the namespace of their enclosing namespace! This is consistent to what we have found out in the previous subchapter: A variable defined inside of a function is local unless it is explicitly marked as global. In other words, we can refer a variable name in any enclosing scope, but we can only rebind variable names in the local scope by assigning to it or in the module-global scope by using a global declaration. We need a way to access variables of other scopes as well. The way to do this are nonlocal definitions, which we will explain in the next chapter.

NONLOCAL VARIABLES

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. One difference to global variables lies in the fact that it is not possible to change variables from the module scope, i.e. variables which are not defined inside of a function, by using the nonlocal statement. We show this in the two following examples:

```
def f():
    global x
    print(x)

x = 3
f()
```

This program is correct and returns the number 3 as the output. We will change "global" to "nonlocal" in the following program:

```
def f():
    nonlocal x
    print(x)

x = 3
f()
```

The program, which we have saved as example1.py, cannot be executed anymore now. We get the following error:

```
File "example1.py", line 2
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found
```

This means that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

```
def f():
    x = 42
    def g():
        nonlocal x
        x = 43
        print("Before calling g: " + str(x))
        print("Calling g now:")
        g()
        print("After calling g: " + str(x))

    x = 3
    f()
    print("x in main: " + str(x))
```

Calling the previous program results in the following output:

```
Before calling g: 42
Calling g now:
After calling g: 43
x in main: 3
```

In the previous example the variable x was defined prior to the call of g. We get an error if it isn't defined:

```
def f():
    #x = 42
```

```

def g():
    nonlocal x
    x = 43
print("Before calling g: " + str(x))
print("Calling g now:")
g()
print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))

```

We get the following error message:

```

File "example3.py", line 4
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found

```

The program works fine - with or without the line "x = 42" inside of f - , when we change "nonlocal" to "global":

```

def f():
    #x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))

```

This leads to the following output:

```

Calling g now:
The value of x after the call to g: 43

Before calling g: 3
Calling g now:
After calling g: 43
x in main: 43

```

Yet there is a huge difference: The value of the global x is changed now!

DECORATORS

INTRODUCTION

Decorators belong most probably to the most beautiful and most powerful design possibilities in Python, but at the same time the concept is considered by many as complicated to get into. To be precise, the usage of decorators is very easy, but writing decorators can be complicated, especially if you are not experienced with decorators and some functional programming concepts.

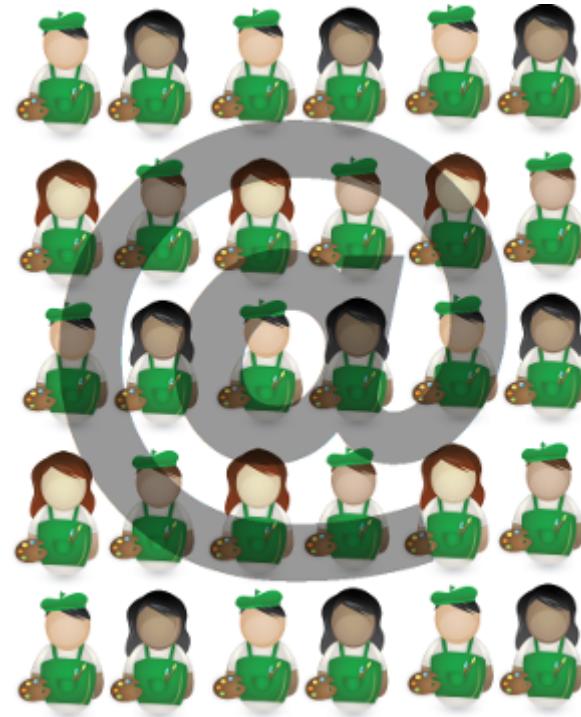
Even though it is the same underlying concept, we have two different kinds of decorators in Python:

- Function decorators
- Class decorators

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

You may also consult our chapter on [memoization with decorators](#).

If you like the image on the right side of this page and if you are also interested in image processing with Python, Numpy, Scipy and Matplotlib, you will definitely like our chapter on [Image Processing Techniques](#), it explains the whole process of the making-of of our decorator and at sign picture!



FIRST STEPS TO DECORATORS

We know from our various Python training classes that there are some sticking points in the definitions of decorators, where many beginners get stuck.

Therefore, we will introduce decorators by repeating some important aspects of

functions. First you have to know or remember that function names are references to functions and that we can assign multiple names to the same function:

```
>>> def succ(x):
...     return x + 1
...
>>> successor = succ
>>> successor(10)
11
>>> succ(10)
11
```

This means that we have two names, i.e. "succ" and "successor" for the same function. The next important fact is that we can delete either "succ" or "successor" without deleting the function itself.

```
>>> del succ
>>> successor(10)
11
```

FUNCTIONS INSIDE FUNCTIONS

The concept of having or defining functions inside of a function is completely new to C or C++ programmers:

```
def f():
    def g():
        print("Hi, it's me 'g'")
        print("Thanks for calling me")

        print("This is the function 'f'")
        print("I am calling 'g' now:")
        g()

f()
```

We will get the following output, if we start the previous program:

```
This is the function 'f'
I am calling 'g' now:
Hi, it's me 'g'
Thanks for calling me
```

Another example using "proper" return statements in the functions:

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32
```

```

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

print(temperature(20))

```

The output:

```
It's 68.0 degrees!
```

FUNCTIONS AS PARAMETERS

If you solely look at the previous examples, this doesn't seem to be very useful. It gets useful in combination with two further powerful possibilities of Python functions. Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

```

def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)

```

The output looks like this:

```
Hi, it's me 'f'
I will call 'func' now
Hi, it's me 'g'
Thanks for calling me
```

You may not be satisfied with the output. 'f' should write that it calls 'g' and not 'func'. Of course, we need to know what the 'real' name of func is. For this purpose, we can use the attribute `__name__`, as it contains this name:

```

def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)

```

```
f(g)
```

The output explains once more what's going on:

```
Hi, it's me 'f'  
I will call 'func' now  
Hi, it's me 'g'  
Thanks for calling me  
func's real name is g
```

Another example:

```
import math

def foo(func):
    print("The function " + func.__name__ + " was passed to foo")
    res = 0
    for x in [1, 2, 2.5]:
        res += func(x)
    return res

print(foo(math.sin))
print(foo(math.cos))
```

The previous example returns the following output

```
The function sin was passed to foo
2.3492405557375347
The function cos was passed to foo
-0.6769881462259364
```

FUNCTIONS RETURNING FUNCTIONS

The output of a function is also a reference to an object. Therefore functions can return references to function objects.

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

The previous example returns the following output:

5
7

We will implement a polynomial "factory" function now. We will start with writing a version which can create polynomials of degree 2.

$$p(x) = a \cdot x^2 + b \cdot x + c$$

The Python implementation as a polynomial factory function can be written like this:

```
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x**2 + b * x + c
    return polynomial

p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

We can generalize our factory function so that it can work for polynomials of arbitrary degree:

$$\sum_{k=0}^n a_k \cdot x^k = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \dots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

```
def polynomial_creator(*coefficients):
    """ coefficients are in the form a_0, a_1, ... a_n
    """
    def polynomial(x):
        res = 0
        for index, coeff in enumerate(coefficients):
            res += coeff * x** index
        return res
    return polynomial

p1 = polynomial_creator(4)
p2 = polynomial_creator(2, 4)
p3 = polynomial_creator(2, 3, -1, 8, 1)
p4 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x), p3(x), p4(x))
```

The function p3 implements, for example, the following polynomial:

$$p_3(x) = x^4 + 8 \cdot x^3 - x^2 + 3 \cdot x + 2$$

If we start this program we get the following output:

```
(-2, 4, -6, -56, -1)
(-1, 4, -2, -9, -2)
(0, 4, 2, 2, -1)
(1, 4, 6, 13, 2)
```

If you want to learn more about polynomials and how to create a polynomial class, you can continue with our chapter on [Polynomials](#).

A SIMPLE DECORATOR

Now we have everything ready to define our first simple decorator:

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

If you look at the following output of the previous program, you can see what's going on.
After the decoration "foo = our_decorator(foo)", foo is a reference to the 'function_wrapper'. 'foo' will be called inside of 'function_wrapper', but before and after the call some additional code will be executed, i.e. in our case two print functions.

```
We call foo before decoration:
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 42
After calling foo
```

THE USUAL SYNTAX FOR DECORATORS IN PYTHON

The decoration in Python is usually not performed in the way we did it in our previous example, even though the notation `foo = our_decorator(foo)` is catchy and easy to grasp. This is the reason, why we used it! You can also see a design problem in our previous approach. "foo" existed in the same program in two versions, before decoration and after decoration.

We will do a proper decoration now. The decoration occurs in the line before the function header. The "`@`" is followed by the decorator function name.

We will rewrite now our initial example. Instead of writing the statement

```
foo = our_decorator(foo)
```

we can write

```
@our_decorator
```

But this line has to be directly positioned in front of the decorated function. The complete example looks like this now:

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

We can decorate every other function which takes one parameter with our decorator '`our_decorator`'. We demonstrate this in the following. We have slightly changed our function wrapper, so that we can see the result of the function calls:

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def succ(n):
    return n + 1
```

```
succ(10)
```

The output of the previous program:

```
Before calling succ
11
After calling succ
```

It is also possible to decorate third party functions, e.g. functions we import from a module. We can't use the Python syntax with the "at" sign in this case:

```
from math import sin, cos

def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

We get the following output:

```
Before calling sin
9.265358966049026e-05
After calling sin
Before calling cos
-0.9999999957076562
After calling cos
```

Summarizing we can say that a decorator in Python is a callable Python object that is used to modify a function, method or class definition. The original object, the one which is going to be modified, is passed to a decorator as an argument. The decorator returns a modified object, e.g. a modified function, which is bound to the name used in the definition.

The previous function_wrapper works only for functions with exactly one parameter. We provide a generalized version of the function_wrapper, which accepts functions with arbitrary parameters in the following example:

```
from random import random, randint, choice

def our_decorator(func):
    def function_wrapper(*args, **kwargs):
        print("Before calling " + func.__name__)
        res = func(*args, **kwargs)
```

```

        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

random = our_decorator(random)
randint = our_decorator(randint)
choice = our_decorator(choice)

random()
randint(3, 8)
choice([4, 5, 6])

```

The result looks as expected:

```

Before calling random
0.16420183945821654
After calling random
Before calling randint
8
After calling randint
Before calling choice
5
After calling choice

```

USE CASES FOR DECORATORS

CHECKING ARGUMENTS WITH A DECORATOR

In our chapter about recursive functions we introduced the factorial function. We wanted to keep the function as simple as possible and we didn't want to obscure the underlying idea, so we hadn't incorporated any argument checks. So, if somebody had called our function with a negative argument or with a float argument, our function would have got into an endless loop.

The following program uses a decorator function to ensure that the argument passed to the function factorial is a positive integer:

```

def argument_test_natural_number(f):
    def helper(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1

```

```

        else:
            return n * factorial(n-1)

for i in range(1,10):
    print(i, factorial(i))

print(factorial(-1))

```

COUNTING FUNCTION CALLS WITH DECORATORS

The following example uses a decorator to count the number of times a function has been called. To be precise, we can use this decorator solely for functions with exactly one parameter:

```

def call_counter(func):
    def helper(x):
        helper.calls += 1
        return func(x)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

print(succ.calls)
for i in range(10):
    succ(i)

print(succ.calls)

```

The output looks like this:

```

0
10

```

We pointed out that we can use our previous decorator only for functions, which take exactly one parameter. We will use the `*args` and `**kwargs` notation to write decorators which can cope with functions with an arbitrary number of positional and keyword parameters.

```

def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0

    return helper

@call_counter

```

```

def succ(x):
    return x + 1

@call_counter
def mull(x, y=1):
    return x*y + 1

print(succ.calls)
for i in range(10):
    succ(i)
mull(3, 4)
mull(4)
mull(y=3, x=2)

print(succ.calls)
print(mull.calls)

```

The output looks like this:

```

0
10
3

```

DECORATORS WITH PARAMETERS

We define two decorators in the following code:

```

def evening_greeting(func):
    def function_wrapper(x):
        print("Good evening, " + func.__name__ + " returns:")
        func(x)
    return function_wrapper

def morning_greeting(func):
    def function_wrapper(x):
        print("Good morning, " + func.__name__ + " returns:")
        func(x)
    return function_wrapper

@evening_greeting
def foo(x):
    print(42)

foo("Hi")

```

These two decorators are nearly the same, except for the greeting. We want to add a parameter to the decorator to be capable of customizing the greeting, when we do the decoration. We have to wrap another function around our previous decorator function to accomplish this. We can now easily say "Good Morning" in the Greek way:

```

def greeting(expr):
    def greeting_decorator(func):
        def function_wrapper(x):
            print(expr + ", " + func.__name__ + " returns:")
            func(x)
        return function_wrapper
    return greeting_decorator

@greeting("καλημερα")
def foo(x):
    print(42)

foo("Hi")

```

The output:

```

καλημερα, foo returns:
42

```

If we don't want or cannot use the "at" decorator syntax, we can do it with function calls:

```

def greeting(expr):
    def greeting_decorator(func):
        def function_wrapper(x):
            print(expr + ", " + func.__name__ + " returns:")
            func(x)
        return function_wrapper
    return greeting_decorator

def foo(x):
    print(42)

greeting2 = greeting("καλημερα")
foo = greeting2(foo)
foo("Hi")

```

The result is the same as before:

```

καλημερα, foo returns:
42

```

Of course, we don't need the additional definition of "greeting2". We can directly apply the result of the call "greeting("καλημερα")" on "foo":

```

foo = greeting("καλημερα") (foo)

```

USING WRAPS FROM FUNCTOOLS

The way we have defined decorators so far hasn't taken into account that the attributes

- `__name__` (name of the function),
- `__doc__` (the docstring) and
- `__module__` (The module in which the function is defined)

of the original functions will be lost after the decoration.

The following decorator will be saved in a file greeting_decorator.py:

```
def greeting(func):
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper
```

We call it in the following program:

```
from greeting_decorator import greeting

@greeting
def f(x):
    """ just some silly function """
    return x + 4

f(10)
print("function name: " + f.__name__)
print("docstring: " + f.__doc__)
print("module name: " + f.__module__)
```

We get the following "unwanted" results:

```
Hi, f returns:
function name: function_wrapper
docstring: function_wrapper of greeting
module name: greeting_decorator
```

We can save the original attributes of the function f, if we assign them inside of the decorator. We change our previous decorator accordingly and save it as greeting_decorator_manually.py:

```
def greeting(func):
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
        function_wrapper.__name__ = func.__name__
        function_wrapper.__doc__ = func.__doc__
        function_wrapper.__module__ = func.__module__
    return function_wrapper
```

In our main program, all we have to do is change the import statement to

```
from greeting_decorator_manually import greeting
```

Now we get the proper results:

```
Hi, f returns:
function name: f
docstring: just some silly function
module name: __main__
```

Fortunately, we don't have to add all this code to our decorators to have these results. We can import the decorator "wraps" from functools instead and decorate our function in the decorator with it:

```
from functools import wraps

def greeting(func):
    @wraps(func)
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper
```

CLASSES INSTEAD OF FUNCTIONS

THE __CALL__ METHOD

So far we used functions as decorators. Before we can define a decorator as a class, we have to introduce the `__call__` method of classes. We mentioned already that a decorator is simply a callable object that takes a function as an input parameter. A function is a callable object, but lots of Python programmers don't know that there are other callable objects. A callable object is an object which can be used and behaves like a function but might not be a function. It is possible to define classes in a way that the instances will be callable objects. The `__call__` method is called, if the instance is called "like a function", i.e. using brackets.

```
class A:

    def __init__(self):
        print("An instance of A was initialized")

    def __call__(self, *args, **kwargs):
        print("Arguments are:", args, kwargs)

x = A()
print("now calling the instance:")
x(3, 4, x=11, y=10)
print("Let's call it again:")
x(3, 4, x=11, y=10)
```

We get the following output:

```
An instance of A was initialized
now calling the instance:
Arguments are: (3, 4) {'x': 11, 'y': 10}
Let's call it again:
Arguments are: (3, 4) {'x': 11, 'y': 10}
```

We can write a class for the fibonacci function by using the `__call__` method:

```
class Fibonacci:

    def __init__(self):
        self.cache = {}

    def __call__(self, n):
        if n not in self.cache:
            if n == 0:
                self.cache[0] = 0
            elif n == 1:
                self.cache[1] = 1
            else:
                self.cache[n] = self.__call__(n-1) +
self.__call__(n-2)
        return self.cache[n]

fib = Fibonacci()

for i in range(15):
    print(fib(i), end=", ")
```

The output:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
```

You can find further information on the `__class__` method in the chapter **Magic Functions** of our tutorial.

USING A CLASS AS A DECORATOR

We will rewrite the following decorator as a class:

```
def decorator1(f):
    def helper():
        print("Decorating", f.__name__)
        f()
    return helper

@decorator1
def foo():
    print("inside foo()")

foo()
```

The following decorator implemented as a class does the same "job":

```
class decorator2:

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```

Both versions return the same output:

```
Decorating foo
inside foo()
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

MEMOIZATION WITH DECORATORS

DEFINITION OF MEMOIZATION

The term "memoization" was introduced by Donald Michie in the year 1968. It's based on the Latin word memorandum, meaning "to be remembered". It's not a misspelling of the word memorization, though in a way it has something in common. Memoisation is a technique used in computing to speed up programs. This is accomplished by memorizing the calculation results of processed input such as the results of function calls. If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided. In many cases a simple array is used for storing the results, but lots of other structures can be used as well, such as associative arrays, called hashes in Perl or dictionaries in Python.



Memoization can be explicitly programmed by the programmer, but some programming languages like Python provide mechanisms to automatically memoize functions.

MEMOIZATION WITH FUNCTION DECORATORS

You may consult our chapter on [decorators](#) as well. Especially, if you may have problems in understanding our reasoning.

In our previous chapter about [recursive functions](#), we worked out an iterative and a recursive version to calculate the Fibonacci numbers. We have shown that a direct implementation of the mathematical definition into a recursive function like the following has an exponential runtime behaviour:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

We also presented a way to improve the runtime behaviour of the recursive version by

adding a dictionary to memorize previously calculated values of the function. This is an example of explicitly using the technique of memoization, but we didn't call it like this. The disadvantage of this method is that the clarity and the beauty of the original recursive implementation is lost.

The "problem" is that we changed the code of the recursive fib function. The following code doesn't change our fib function, so that its clarity and legibility isn't touched. To this purpose, we define and use a function which we call memoize. memoize() takes a function as an argument. The function memoize uses a dictionary "memo" to store the function results. Though the variable "memo" as well as the function "f" are local to memoize, they are captured by a closure through the helper function which is returned as a reference by memoize(). So, the call memoize(fib) returns a reference to the helper() which is doing what fib() would do on its own plus a wrapper which saves the calculated results. For an integer 'n' fib(n) will only be called, if n is not in the memo dictionary. If it is in it, we can output memo[n] as the result of fib(n).

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib = memoize(fib)

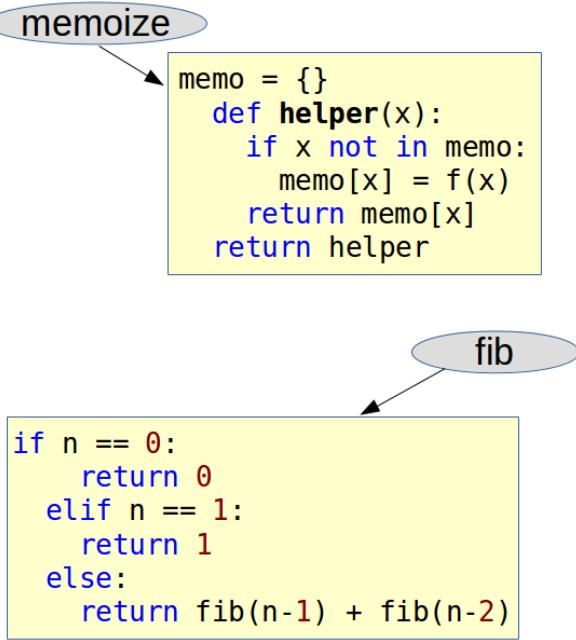
print(fib(40))
```

Let's look at the line in our code where we call memoize with fib as the argument:

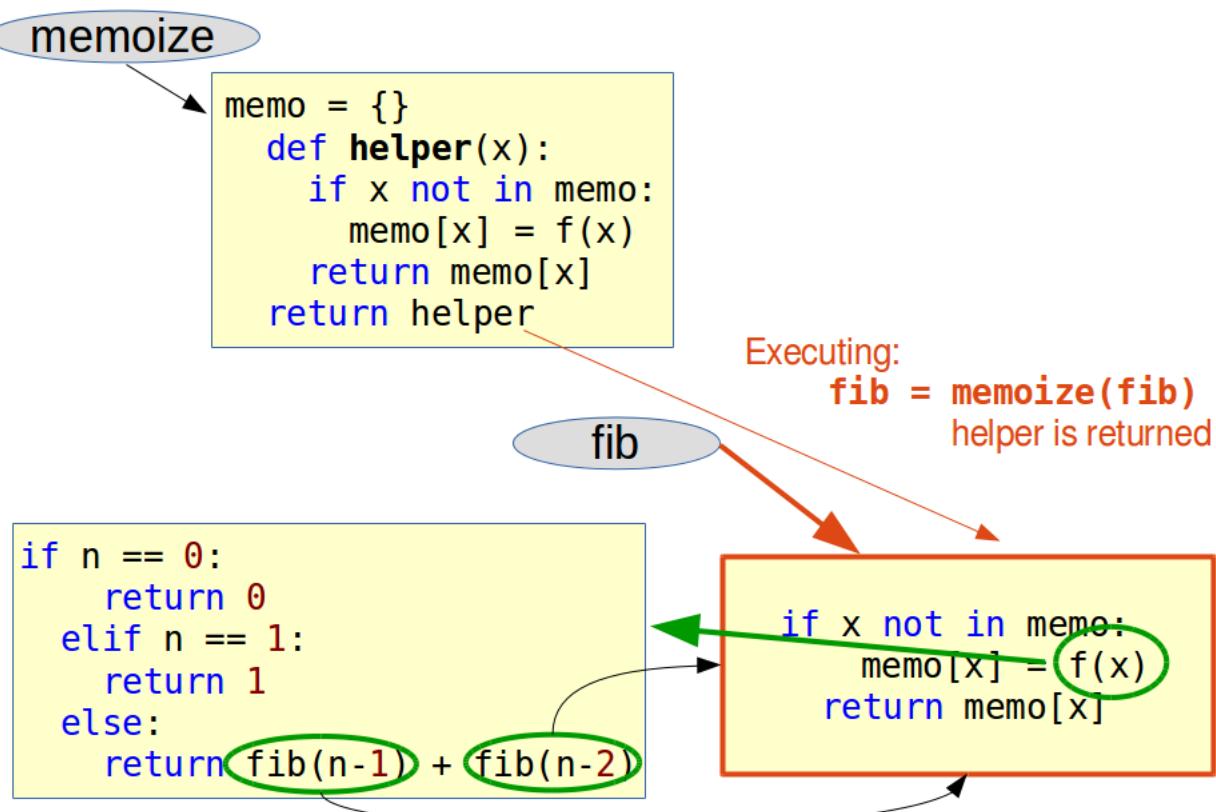
```
fib = memoize(fib)
```

Doing this, we turn memoize into a decorator. One says that the fib function is decorated by the memoize() function.

We will illustrate with the following diagrams how the decoration is accomplished. The first diagram illustrates the state before the decoration, i.e. before we call `fib = memoize(fib)`. We can see the function names referencing their bodies:



After having executed `fib = memoize(fib)` `fib` points to the body of the helper function, which had been returned by `memoize`. We can also perceive that the code of the original `fib` function can from now on only be reached via the "f" function of the helper function. There is no other way anymore to call the original `fib` directly, i.e. there is no other reference to it. The decorated Fibonacci function is called in the return statement `return fib(n-1) + fib(n-2)`, this means the code of the helper function which had been returned by `memoize`:



Another point in the context of decorators deserves special mention: We don't usually write a decorator for just one use case or function. We rather use it multiple times for different functions. So we could imagine having further functions func1, func2, func3 and so on, which consume also a lot of time. Therefore, it makes sense to decorate each one with our decorator function "memoize":

```
fib = memoize(fib)
func1 = memoize(func1)
func2 = memoize(func2)
func3 = memoize(func3)
# and so on
```

We haven't used the Pythonic way of writing a decorator. Instead of writing the statement

```
fib = memoize(fib)
```

we should have "decorated" our fib function with:

```
@memoize
```

But this line has to be directly in front of the decorated function, in our example fib(). The complete example in a Pythonic way looks like this now:

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

@memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(40))
```

USING A CALLABLE CLASS FOR MEMOIZATION

This subchapter can be skipped without problems by those who don't know about object orientation so far.

We can encapsulate the caching of the results in a class as well, as you can see in the following example:

```
class Memoize:

    def __init__(self, fn):
        self.fn = fn
        self.memo = {}

    def __call__(self, *args):
        if args not in self.memo:
            self.memo[args] = self.fn(*args)
        return self.memo[args]

@Memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

As we are using a dictionary, we can't use mutable arguments, i.e. the arguments have to be immutable.

EXERCISE

1. Our exercise is an old riddle, going back to 1612. The French Jesuit Claude-Gaspar Bachet phrased it. We have to weigh quantities (e.g. sugar or flour) from 1 to 40 pounds. What is the least number of weights that can be used on a balance scale to way any of these quantities.

The first idea might be to use weights of 1, 2, 4, 8, 16 and 32 pounds. This is a minimal number, if we restrict ourself to put weights on one side and the stuff, e.g. the sugar, on the other side. But it is possible to put weights on both pans of the scale. Now, we need only four weights, i.e. 1, 3, 9, 27



Write a Python function `weigh()`, which calculates the weights needed and their distribution on the pans to weigh any amount from 1 to 40.

SOLUTION

1. We need the function `linear_combination()` from our chapter "[Linear Combinations](#)".

```
def factors_set():
    factors_set = ( (i, j, k, l) for i in [-1, 0, 1]
                    for j in [-1, 0, 1]
                    for k in [-1, 0, 1]
                    for l in [-1, 0, 1])
    for factor in factors_set:
        yield factor

def memoize(f):
    results = {}
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper

@memoize
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27      """
    weighs = (1, 3, 9, 27)

    for factors in factors_set():
        sum = 0
        for i in range(len(factors)):
            sum += factors[i] * weighs[i]
        if sum == n:
            return factors
```

With this, it is easy to write our function `weigh()`.

```
def weigh(pounds):
    weights = (1, 3, 9, 27)
    scalars = linear_combination(pounds)
    left = ""
    right = ""
    for i in range(len(scalars)):
        if scalars[i] == -1:
            left += str(weights[i]) + " "
        elif scalars[i] == 1:
            right += str(weights[i]) + " "
    return (left,right)

for i in [2, 3, 4, 7, 8, 9, 20, 40]:
    pans = weigh(i)
```

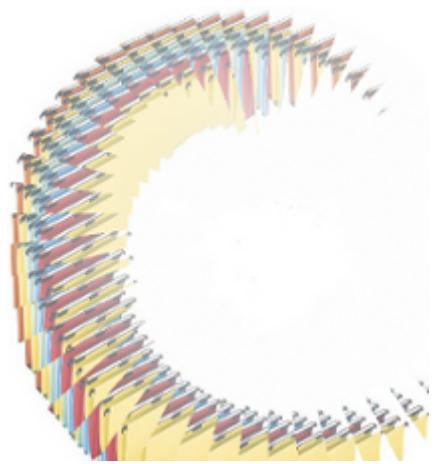
```
print("Left pan: " + str(i) + " plus " + pans[0])
print("Right pan: " + pans[1] + "\n")
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

FILE MANAGEMENT

FILES IN GENERAL

It's hard to find anyone in the 21st Century, who doesn't know what a file is. If we say file, we mean of course, a file on a computer. There may be people who don't know anymore the "container", like a cabinet or a folder, for keeping papers archived in a convenient order. A file on a computer is the modern counterpart of this. It is a collection of information, which can be accessed and used by a computer program. Usually, a file resides on a durable storage. Durable means that the data is persistent, i.e. it can be used by other programs after the program which has created or manipulated it has terminated.



The term file management in the context of computers refers to the manipulation of data in a file or files and documents on a computer. Though everybody has an understanding of the term file, we present a formal definition anyway:

A file or a computer file is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

The term "file" - as we have described it in the previous paragraph - appeared in the history of computers very early. Usage can be tracked down to the year 1952, when punch cards were used.

A programming language without the capability to store and retrieve previously stored information would be hardly useful.

The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.

READING AND WRITING FILES IN PYTHON

The syntax for reading and writing files in Python is similar to programming languages like C, C++, Java, Perl, and others but a lot easier to handle.

In our first example we want to show how to read data from a file. The way of telling Python that we want to read from a file is to use the open function. The first parameter is the name of the file we want to read and with the second parameter, assigned to the value "r", we state that we want to read from the file:

```
fobj = open("ad_lebiam.txt", "r")
```

The "r" is optional. An open() command with just a file name is opened for reading per default. The open() function returns a file object, which offers attributes and methods.

```
fobj = open("ad_lebiam.txt")
```

After we have finished working with a file, we have to close it again by using the file object method close():

```
fobj.close()
```

Now we want to finally open and read a file. The method rstrip() in the following example is used to strip off whitespaces (newlines included) from the right side of the string "line":

```
fobj = open("ad_lebiam.txt")
for line in fobj:
    print(line.rstrip())
fobj.close()
```

If we save this script and call it, we get the following output, provided that the text file "ad_lebiam.txt" is available:

```
$ python file_read.py
V. ad Lesbia

VIVAMUS mea Lesbia, atque amemus,
rumoresque senum severiorum
omnes unius aestimemus assis!
soles occidere et redire possunt:
nobis cum semel occidit breuis lux,
nox est perpetua una dormienda.
da mi basia mille, deinde centum,
dein mille altera, dein secunda centum,
deinde usque altera mille, deinde centum.
dein, cum milia multa fecerimus,
conturbabimus illa, ne sciamus,
aut ne quis malus inuidere possit,
cum tantum sciat esse basiorum.
(GAIUS VALERIUS CATULLUS)
```

By the way, the poem above is a love poem of the ancient Roman poet Catull, who was hopelessly in love with a woman called Lesbia.

WRITE INTO A FILE

Writing to a file is as easy as reading from a file. To open a file for writing we set the second parameter to "w" instead of "r". To actually write the data into this file, we use the method `write()` of the file handle object.

Let's start with a very simple and straightforward example:

```
fh = open("example.txt", "w")
fh.write("To write or not to write\nthat is the question!\n")
fh.close()
```

Especially if you are writing to a file, you should never forget to close the file handle again. Otherwise you will risk to end up in a non consistent state of your data.

You will often find the `with` statement for reading and writing files. The advantage is that the file will be automatically closed after the indented block after the `with` has finished execution:

```
with open("example.txt", "w") as fh:
    fh.write("To write or not to write\nthat is the question!\n")
```

Our first example can also be rewritten like this with the `with` statement:

```
with open("ad_lesbiam.txt") as fobj:
    for line in fobj:
        print(line.rstrip())
```

Example for simultaneously reading and writing:

```
fobj_in = open("ad_lesbiam.txt")
fobj_out = open("ad_lesbiam2.txt", "w")
i = 1
for line in fobj_in:
    print(line.rstrip())
    fobj_out.write(str(i) + ": " + line)
    i = i + 1
fobj_in.close()
fobj_out.close()
```

Every line of the input text file is prefixed by its line number. So the result looks like this:

```
$ more ad_lesbiam2.txt
1: V. ad Lesbian
```

```

2:
3: VIVAMUS mea Lesbia, atque amemus,
4: rumoresque senum severiorum
5: omnes unius aestimemus assis!
6: soles occidere et redire possunt:
7: nobis cum semel occidit breuis lux,
8: nox est perpetua una dormienda.
9: da mi basia mille, deinde centum,
10: dein mille altera, dein secunda centum,
11: deinde usque altera mille, deinde centum.
12: dein, cum milia multa fecerimus,
13: conturbabimus illa, ne sciamus,
14: aut ne quis malus inuidere possit,
15: cum tantum sciat esse basiorum.
16: (GAIUS VALERIUS CATULLUS)

```

There is one possible problem, which we have to point out: What happens if we open a file for writing, and this file already exists. You can consider yourself fortunate, if the content of this file was of no importance, or if you have a backup of it. Otherwise you have a problem, because as soon as an `open()` with a "w" has been executed the file will be removed. This is often what you want, but sometimes you just want to append to the file, like it's the case with logfiles.

If you want to append something to an existing file, you have to use "a" instead of "w".

READING IN ONE GO

So far we worked on files line by line by using a for loop. Very often, especially if the file is not too large, it's more convenient to read the file into a complete data structure, e.g. a string or a list. The file can be closed after reading and the work is accomplished on this data structure:

```

>>> poem = open("ad_lesbiam.txt").readlines()
>>> print(poem)
['V. ad Lesbiam \n', '\n', 'VIVAMUS mea Lesbia, atque amemus,\n',
 'rumoresque senum severiorum\n', 'omnes unius aestimemus assis!\n',
 'soles occidere et redire possunt:\n', 'nobis cum semel occidit
 breuis lux,\n', 'nox est perpetua una dormienda.\n', 'da mi basia
 mille, deinde centum,\n', 'dein mille altera, dein secunda
 centum,\n', 'deinde usque altera mille, deinde centum.\n', 'dein,
 cum milia multa fecerimus,\n', 'conturbabimus illa, ne sciamus,\n',
 'aut ne quis malus inuidere possit,\n', 'cum tantum sciat esse
 basiorum.\n', '(GAIUS VALERIUS CATULLUS)']
>>> print(poem[2])
VIVAMUS mea Lesbia, atque amemus,

```

In the above example, the complete poem is read into the list `poem`. We can access e.g. the 3rd line with `poem[2]`.

Another convenient way to read in a file might be the method `read()` of `open`. With this method we can read the complete file into a string, as we can see in the next example:

```
>>> poem = open("ad_lebian.txt").read()
>>> print(poem[16:34])
VIVAMUS mea Lesbia
>>> type(poem)
<type 'str'>
>>>
```

This string contains the complete content of the file, which includes the carriage returns and line feeds.

RESETTING THE FILES CURRENT POSITION

It's possible to set - or reset - a file's position to a certain position, also called the offset. To do this, we use the method `seek`. It has only one parameter in Python3 (no "whence" is available as in Python2). The parameter of `seek` determines the offset which we want to set the current position to. To work with `seek`, we will often need the method `tell`, which "tells" us the current position. When we have just opened a file, it will be zero. We will demonstrate the way of working with both `seek` and `tell` in the following example. You have to create a file called "buck_mulligan.txt" with the content "Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed.":

```
>>> fh = open("buck_mulligan.txt")
>>> fh.tell()
0
>>> fh.read(7)
'Stately'
>>> fh.tell()
7
>>> fh.read()
', plump Buck Mulligan came from the stairhead, bearing a bowl
of\nlather on which a mirror and a razor lay crossed.\n'
>>> fh.tell()
122
>>> fh.seek(9)
9
>>> fh.read(5)
'plump'
```

It's also possible to set the file position relative to the current position by using `tell` correspondingly:

```
>>> fh = open("buck_mulligan.txt")
>>> fh.read(15)
'Stately, plump '
>>> # set the current position 6 characters to the left:
...
>>> fh.seek(fh.tell() - 6)
```

```

9
>>> fh.read(5)
'plump'
>>> # now, we will advance 29 characters to the
>>> # 'right' relative to the current position:
...
>>> fh.seek(fh.tell() + 29)
43
>>> fh.read(10)
'stairhead,'

>>>

```

READ AND WRITE TO THE SAME FILE

In the following example we will open a file for reading and writing at the same time. If the file doesn't exist, it will be created. If you want to open an existing file for read and write, you should better use "r+", because this will not delete the content of the file.

```

fh = open('colours.txt', 'w+')
fh.write('The colour brown')

# Go to the 12th byte in the file, counting starts with 0
fh.seek(11)
print(fh.read(5))
print(fh.tell())
fh.seek(11)
fh.write('green')
fh.seek(0)
content = fh.read()
print(content)

```

We get the following output:

```

brown
16
The colour green

```

"HOW TO GET INTO A PICKLE"

We don't mean what the heading says. On the contrary, we want to prevent any nasty situation, like loosing the data, which your Python program has calculated. So, we will show you, how you can save your data in an easy way that you or better your program can reread them at a later date again. We are "pickling" the data, so that nothing gets lost.

Python offers for this purpose a module, which is called "pickle". With the algorithms of

the pickle module we can serialize and de-serialize Python object structures. "Pickling" denotes the process which converts a Python object hierarchy into a byte stream, and "unpickling" on the other hand is the inverse operation, i.e. the byte stream is converted back into an object hierarchy. What we call pickling (and unpickling) is also known as "serialization" or "flattening" a data structure.

An object can be dumped with the `dump` method of the pickle module:

```
pickle.dump(obj, file[, protocol, *, fix_imports=True])
```

`dump()` writes a pickled representation of `obj` to the open file object `file`. The optional `protocol` argument tells the pickler to use the given protocol:

- Protocol version 0 is the original (before Python3) human-readable (ascii) protocol and is backwards compatible with previous versions of Python
- Protocol version 1 is the old binary format which is also compatible with previous versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.
- Protocol version 3 was introduced with Python 3.0. It has explicit support for bytes and cannot be unpickled by Python 2.x pickle modules. It's the recommended protocol of Python 3.x.

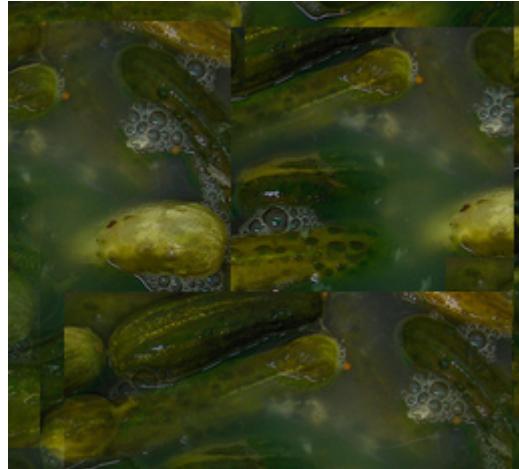
The default protocol of Python3 is 3.

If `fix_imports` is True and `protocol` is less than 3, pickle will try to map the new Python3 names to the old module names used in Python2, so that the pickle data stream is readable with Python 2.

Objects which have been dumped to a file with `pickle.dump` can be reread into a program by using the method `pickle.load(file)`. `pickle.load` recognizes automatically, which format had been used for writing the data.

A simple example:

```
>>> import pickle
>>>
>>> cities = ["Paris", "Dijon", "Lyon", "Strasbourg"]
>>> fh = open("data.pkl", "bw")
>>> pickle.dump(cities, fh)
>>> fh.close()
```



The file data.pkl can be read in again by Python in the same or another session or by a different program:

```
>>> import pickle
>>> f = open("data.pkl", "rb")
>>> villes = pickle.load(f)
>>> print(villes)
['Paris', 'Dijon', 'Lyon', 'Strasbourg']
>>>
```

Only the objects and not their names are saved. That's why we use the assignment to villes in the previous example, i.e. data = pickle.load(f).

In our previous example, we had pickled only one object, i.e. a list of French cities. But what about pickling multiple objects? The solution is easy: We pack the objects into another object, so we will only have to pickle one object again. We will pack two lists "programming_languages" and "python_dialects" into a list pickle_object in the following example:

```
>>> import pickle
>>> fh = open("data.pkl", "bw")
>>> programming_languages = ["Python", "Perl", "C++", "Java",
    "Lisp"]
>>> python_dialects = ["Jython", "IronPython", "CPython"]
>>> pickle_object = (programming_languages, python_dialects)
>>> pickle.dump(pickle_object, fh)
>>> fh.close()
```

The pickled data from the previous example, - i.e. the data which we have written to the file data.pkl, - can be separated into two lists again, when we read back in again the data:

```
>>> import pickle
>>> f = open("data.pkl", "rb")
>>> (languages, dialects) = pickle.load(f)
>>> print(languages, dialects)
['Python', 'Perl', 'C++', 'Java', 'Lisp'] ['Jython', 'IronPython',
'CPython']
>>>
```

SHELVE MODULE

One drawback of the pickle module is that it is only capable of pickling one object at the time, which has to be unpickled in one go. Let's imagine this data object is a dictionary. It may be desirable that we don't have to save and load every time the whole dictionary, but save and load just a single value corresponding to just one key. The shelve module is the solution to this request. A "shelf" - as used in the shelve module - is a persistent, dictionary-like object. The difference with dbm databases is that the values (not the keys!) in a shelf

can be essentially arbitrary Python objects -- anything that the "pickle" module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys have to be strings.

The `shelve` module can be easily used. Actually, it is as easy as using a dictionary in Python. Before we can use a shelf object, we have to import the module. After this, we have to open a shelf object with the `shelve` method `open`. The `open` method opens a special shelf file for reading and writing:

```
>>> import shelve  
>>> s = shelve.open("MyShelve")
```

If the file "MyShelve" already exists, the `open` method will try to open it. If it isn't a shelf file, - i.e. a file which has been created with the `shelve` module, - we will get an error message. If the file doesn't exist, it will be created.

We can use `s` like an ordinary dictionary, if we use strings as keys:

```
>>> s["street"] = "Fleet Str"  
>>> s["city"] = "London"  
>>> for key in s:  
...     print(key)  
...  
city  
street
```

A shelf object has to be closed with the `close` method:

```
>>> s.close()
```

We can use the previously created shelf file in another program or in an interactive Python session:

```
$ python3  
Python 3.2.3 (default, Feb 28 2014, 00:22:33)  
[GCC 4.7.2] on linux2  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>> import shelve  
>>> s = shelve.open("MyShelve")
```

```
>>> s["street"]
'Fleet Str'
>>> s["city"]
'London'
>>>
```

It is also possible to cast a shelf object into an "ordinary" dictionary with the dict function:

```
>>> s
<shelve.DbfilenameShelf object at 0xb7133dcc>
>>> dict(s)
{'city': 'London', 'street': 'Fleet Str'}
>>>
```

The following example uses more complex values for our shelf object:

```
>>> import shelve
>>> tele = shelve.open("MyPhoneBook")
>>> tele["Mike"] = {"first": "Mike", "last": "Miller", "phone": "4689"}
>>> tele["Steve"] = {"first": "Stephan", "last": "Burns",
"phone": "8745"}
>>> tele["Eve"] = {"first": "Eve", "last": "Naomi", "phone": "9069"}
>>> tele["Eve"]["phone"]
'9069'
```

The data is persistent!

To demonstrate this once more, we reopen our MyPhoneBook:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import shelve
>>> tele = shelve.open("MyPhoneBook")
>>> tele["Steve"]["phone"]
'8745'
>>>
```

EXERCISES

1. The file **cities_and_times.txt** contains city names and times. Each line contains the name of the city, followed by the name of the day ("Sun") and the time in the form hh:mm. Read in the file and create an alphabetically ordered list of the form

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)),
 ('Ankara', 'Sun', (10, 52)), ('Athens', 'Sun', (9, 52)),
 ('Atlanta', 'Sun', (2, 52)), ('Auckland', 'Sun', (20, 52)),
 ('Barcelona', 'Sun', (8, 52)), ('Beirut', 'Sun', (9, 52)),
 ...
 ('Toronto', 'Sun', (2, 52)), ('Vancouver', 'Sun', (0, 52)),
 ('Vienna', 'Sun', (8, 52)), ('Warsaw', 'Sun', (8, 52)),
 ('Washington DC', 'Sun', (2, 52)), ('Winnipeg', 'Sun', (1, 52)),
 ('Zurich', 'Sun', (8, 52))]
```

Finally, the list should be dumped for later usage with the pickle module. We will use this list in our chapter on **Numpy dtype**.

SOLUTIONS TO OUR EXERCISES

```
1. import pickle

lines = open("cities_and_times.txt").readlines()
lines.sort()

cities = []
for line in lines:
    *city, day, time = line.split()
    hours, minutes = time.split(":")
    cities.append(( " ".join(city), day, (int(hours),
    int(minutes)) ))

fh = open("cities_and_times.pkl", "bw")
pickle.dump(cities, fh)
```

City names can consist of multiple words like "Salt Lake City". That is why we have to use the asterisk in the line, in which we split a line. So city will be a list with the

words of the city, e.g. ["Salt", "Lake", "City"]. " ".join(city) turns such a list into a "proper" string with the city name, i.e. in our example "Salt Lake City".

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

q

MODULAR PROGRAMMING AND MODULES

MODULAR PROGRAMMING

Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.

There is hardly any product nowadays, which doesn't heavily rely on modularisation, like cars, mobile phones. Computers belong to those products which are modularised to the utmost. So, what's a must for the hardware is an unavoidable necessity for the software running on the computers.



If you want to develop programs which are readable, reliable and maintainable without too much effort, you have to use some kind of modular software design. Especially if your application has a certain size. There exists a variety of concepts to design software in modular form. Modular programming is a software design technique to split your code into separate parts. These parts are called modules. The focus for this separation should be to have modules with no or just few dependencies upon other modules. In other words: Minimization of dependencies is the goal. When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

IMPORTING MODULES

So far we haven't explained what a Python module is. To put it in a nutshell: every file, which has the file extension .py and consists of proper Python code, can be seen or is a module! There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:

```
import math
```

The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:

```
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
>>> math.cos(math.pi/2)
6.123031769111886e-17
>>> math.cos(math.pi)
-1.0
```

It's possible to import more than one module in one import statement. In this case the module names are separated by commas:

```
import math, random
```

import statements can be positioned anywhere in the program, but it's good style to place them directly at the beginning of a program.

If only certain objects of a module are needed, we can import only those:

```
from math import sin, pi
```

The other objects, e.g. cos, are not available after this import. We are capable of accessing sin and pi directly, i.e. without prefixing them with "math."

Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:

```
>>> from math import *
>>> sin(3.01) + tan(cos(2.1)) + e
2.2968833711382604
>>> e
2.718281828459045
>>>
```

It's not recommended to use the asterisk notation in an import statement, except when working in the interactive Python shell. One reason is that the origin of a name can be quite obscure, because it can't be seen from which module it might have been imported. We will demonstrate another serious complication in the following example:

```
>>> from numpy import *
>>> from math import *
>>> print(sin(3))
0.1411200080598672
>>> sin(3)
```

```
0.1411200080598672
>>>
```

Let's slightly change the previous example by changing the order of the imports:

```
>>> from math import *
>>> from numpy import *
>>> print(sin(3))
0.14112000806
>>> sin(3)
0.14112000805986721
>>>
```

People use the asterisk notation, because it is so convenient. It means avoiding a lot of tedious typing. Another way to shrink the typing effort consists in renaming a namespace. A good example for this is the numpy module. You will hardly find an example or a tutorial, in which they will import this module with the statement

```
import numpy
```

It's like an unwritten law to import it with

```
import numpy as np
```

Now you can prefix all the objects of numpy with "np." instead of "numpy.":

```
>>> import numpy as np
>>> np.diag([3, 11, 7, 9])
array([[ 3,  0,  0,  0],
       [ 0, 11,  0,  0],
       [ 0,  0,  7,  0],
       [ 0,  0,  0,  9]])
>>> np.e
2.718281828459045
>>>
```

DESIGNING AND WRITING MODULES

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done, we

just save the following code in the file fibonacci.py:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
def ifib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

The newly created module "fibonacci" is ready for use now. We can import this module like any other module in a program or script. We will demonstrate this in the following interactive Python shell:

```
>>> import fibonacci
>>> fibonacci.fib(7)
13
>>> fibonacci.fib(20)
6765
>>> fibonacci.ifib(42)
267914296
>>> fibonacci.ifib(1000)
43466557686937456435688527675040625802564660517371780402481729089536
55541794905189040387984007925516929592259308032263477520968962323987
33224711616429964409065331879382989696499285160037044761377951668492
28875
>>>
```

Don't try to call the recursive version of the Fibonacci function with large arguments like we did with the iterative version. A value like 42 is already too large. You will have to wait for a long time!

As you can easily imagine: It's a pain if you have to use those functions often in your program and you always have to type in the fully qualified name, i.e. fibonacci.fib(7). One solution consists in assigning a local name to a module function to get a shorter name:

```
>>> fib = fibonacci.ifib
>>> fib(10)
55
>>>
```

But it's better, if you import the necessary functions directly into your module, as we will demonstrate further down in this chapter.

MORE ON MODULES

Usually, modules contain functions or classes, but there can be "plain" statements in them as well. These statements can be used to initialize the module. They are only executed when the module is imported.

Let's look at a module, which only consists of just one statement:

```
print("The module is imported now!")
```

We save with the name "one_time.py" and import it two times in an interactive session:

```
>>> import one_time  
The module is imported now!  
>>> import one_time  
>>>
```

We can see that it was only imported once. Each module can only be imported once per interpreter session or in a program or script. If you change a module and if you want to reload it, you must restart the interpreter again. In Python 2.x, it was possible to reimport the module by using the built-in reload, i.e.reload(modulename):

```
$ python  
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)  
[GCC 4.4.3] on linux2  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>> import one_time  
The module is imported now!  
>>> reload(one_time)  
The module is imported now!  
  
>>>
```

This is not possible anymore in Python 3.x.

You will cause the following error:

```
>>> import one_time  
The module is imported now!  
>>> reload(one_time)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'reload' is not defined  
>>>
```

Since Python 3.0 the reload built-in function has been moved into the imp standard library module. So it's still possible to reload files as before, but the functionality has to be imported. You have to execute an "import imp" and use imp.reload(my_module).

Alternatively, you can use "imp import reload" and use `reload(my_module)`.

Example with reloading the Python3 way:

```
$ python3
Python 3.1.2 (r312:79147, Sep 27 2010, 09:57:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from imp import reload
>>> import one_time
The module is imported now!
>>> reload(one_time)
The module is imported now!

>>>
```

Since version 3.4 you should use the "importlib" module, because `imp.reload` is marked as deprecated:

```
>>> from importlib import reload
>>> import one_time
The module is imported now!
>>> reload(one_time)
The module is imported now!

>>>
```

A module has a private symbol table, which is used as the global symbol table by all functions defined in the module. This is a way to prevent that a global variable of a module accidentally clashes with a user's global variable with the same name. Global variables of a module can be accessed with the same notation as functions, i.e. `modname.name`
A module can import other modules. It is customary to place all import statements at the beginning of a module or a script.

IMPORTING NAMES FROM A MODULE DIRECTLY

Names from a module can directly be imported into the importing module's symbol table:

```
>>> from fibonacci import fib, ifib
>>> ifib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table. It's possible but not recommended to import all names defined in a module, except those beginning with an underscore "_":

```
>>> from fibonacci import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This shouldn't be done in scripts but it's possible to use it in interactive sessions to save typing.

EXECUTING MODULES AS SCRIPTS

Essentially a Python module is a script, so it can be run as a script:

```
python fibo.py
```

The module which has been started as a script will be executed as if it had been imported, but with one exception: The system variable `__name__` is set to "`__main__`". So it's possible to program different behaviour into a module for the two cases. With the following conditional statement the file can be used as a module or as a script, but only if it is run as a script the method `fib` will be started with a command line argument:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

If it is run as a script, we get the following output:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If it is imported, the code in the if block will not be executed:

```
>>> import fibo
>>>
```

RENAMING A NAMESPACE

While importing a module, the name of the namespace can be changed:

```
>>> import math as mathematics
>>> print(mathematics.cos(mathematics.pi))
-1.0
```

After this import there exists a namespace `mathematics` but no namespace `math`. It's possible to import just a few methods from a module:

```
>>> from math import pi,pow as power, sin as sinus
>>> power(2,3)
8.0
>>> sinus(pi)
1.2246467991473532e-16
```

KINDS OF MODULES

There are different kind of modules:

- Those written in Python
They have the suffix: .py
- Dynamically linked C modules
Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter:
It's possible to get a complete list of these modules:

```
import sys
print(sys.builtin_module_names)
```

An error message is returned for Built-in-Modules.

MODULE SEARCH PATH

If you import a module, let's say "import xyz", the interpreter searches for this module in the following locations and in the order given:

1. The directory of the top-level file, i.e. the file being executed.
2. The directories of PYTHONPATH, if this global environment variable of your operating system is set.
3. standard installation path Linux/Unix e.g. in /usr/lib/python3.5.

It's possible to find out where a module is located after it has been imported:

```
>>> import numpy
>>> numpy.__file__
'/usr/lib/python3/dist-packages/numpy/__init__.py'
>>>
>>> import random
>>> random.__file__
'/usr/lib/python3.5/random.py'
>>>
```

The __file__ attribute doesn't always exists. This is the case with modules which are statically linked C libraries.

```
>>> import math
>>> math.__file__
```

```

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

CONTENT OF A MODULE

With the built-in function `dir()` and the name of the module as an argument, you can list all valid attributes and methods for that module.

```

>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__',
 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
 'expm1',
 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
 'gcd',
 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',
 'lgamma',
 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
 'radians',
 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Calling `dir()` without an argument, a list with the names in the current local scope is returned:

```

>>> import math
>>> cities = ["New York", "Toronto", "Berlin", "Washington",
    "Amsterdam", "Hamburg"]
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'cities', 'math']
>>>
```

It's possible to get a list of the Built-in functions, exceptions, and other objects by importing the `builtins` module:

```

>>> import builtins
>>> dir(builtins)
['ArithError', 'AssertionError', 'AttributeError',
 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundException', 'FloatingPointError',
 'FutureWarning',
```

```
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError',
'Runtimewarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit',
'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__build_class__',
'__debug__', '__doc__', '__import__', '__loader__', '__name__',
'__package__', '__spec__',
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable', 'chr',
'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str',
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
>>>
```

PACKAGES

If you have created a lot of modules at some point in time, you may lose the overview about them. You may have dozens or hundreds of modules and they can be categorized into different categories. It is similar to the situation in a file system: Instead of having all files in just one directory, you put them into different ones, being organized according to the topics of the files. We will show in the next chapter of our Python tutorial how to organize modules into packages.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

PACKAGES IN PYTHON

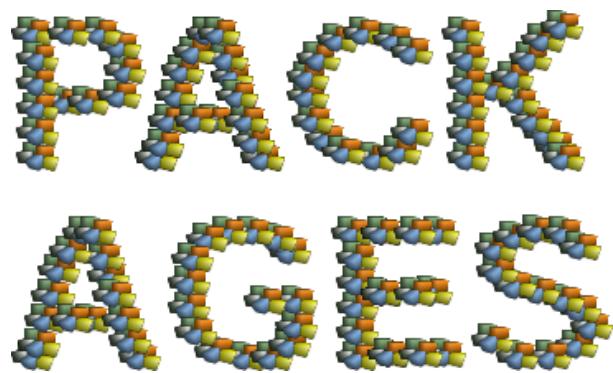
INTRODUCTION

We learned that modules are files containing Python statements and definitions, like function and class definitions. We will learn in this chapter how to bundle multiple modules together to form a package.

A package is basically a directory with Python files and a file with the name `__init__.py`. This means that every directory inside of the Python path, which contains a file named `__init__.py`, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python's module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A, for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different.

A package is imported like a "normal" module.
We will start this chapter with a simple example.



A SIMPLE EXAMPLE

We will demonstrate with a very simple example how to create a package with some Python modules.

First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package "simple_package". This directory needs to contain a file with the name "`__init__.py`". This file can be empty, or it can contain valid Python code. This code will be executed when a package will be imported, so it can be used to initialize a package, e.g. to make sure that some other modules are imported or some values set. Now we can put into this directory all the Python files which will be the submodules of our module.

We create two simple files `a.py` and `b.py` just for the sake of filling the package with

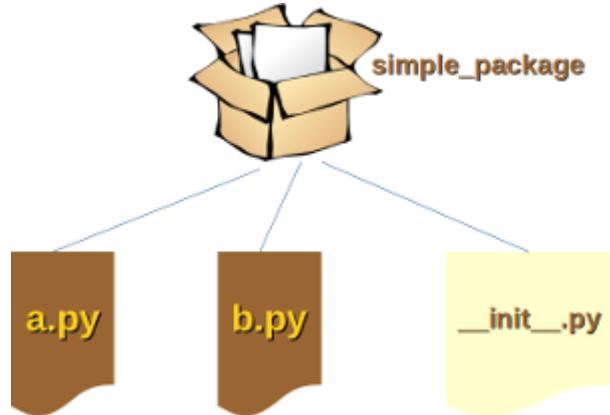
modules.

The content of a.py:

```
def bar():
    print("Hello, function 'bar'
from module 'a' calling")
```

The content of b.py:

```
def foo():
    print("Hello, function 'foo'
from module 'b' calling")
```



We will also add an empty file with the name `__init__.py` inside of `simple_package` directory.

Let's see what's happening, when we import `simple_package` from the interactive Python shell, assuming that the directory `simple_package` is either in the directory from which you call the shell or that it is contained in the search path or environment variable "PYTHONPATH" (from your operating system):

```
>>> import simple_package
>>>
>>> simple_package
<module 'simple_package' from '/home/bernd/Dropbox
(Bodenseo)/websites/python-
course.eu/examples/simple_package/__init__.py'>
>>>
>>> simple_package/a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
>>> simple_package/b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

We can see that the package `simple_package` has been loaded but neither the module "a" nor the module "b"! We can import the modules a and b in the following way:

```
>>> from simple_package import a, b
>>> a.bar()
Hello, function 'bar' from module 'a' calling
>>> b.foo()
Hello, function 'foo' from module 'b' calling
>>>
```

As we have seen at the beginning of the chapter, we can't access neither "a" nor "b" by solely importing `simple_package`.

Yet, there is a way to automatically load these modules. We can use the file `__init__.py` for

this purpose. All we have to do is add the following lines to the so far empty file `__init__.py`:

```
import simple_package.a
import simple_package.b
```

It will work now:

```
>>> import simple_package
>>>
>>> simple_package.a.bar()
Hello, function 'bar' from module 'a' calling
>>>
>>> simple_package.b.foo()
Hello, function 'foo' from module 'b' calling
```

A MORE COMPLEX PACKAGE

We will demonstrate in the following example how we can create a more complex package. We will use the hypothetical sound-Modul which is used in the official tutorial. (see <https://docs.python.org/3/tutorial/modules.html>)

```
sound
| -- effects
|   | -- echo.py
|   | -- __init__.py
|   | -- reverse.py
|   `-- surround.py
| -- filters
|   | -- equalizer.py
|   | -- __init__.py
|   | -- karaoke.py
|   `-- vocoder.py
| -- formats
|   | -- aiffread.py
|   | -- aiffwrite.py
|   | -- auread.py
|   | -- auwrite.py
|   | -- __init__.py
|   | -- wavread.py
|   `-- wavwrite.py
`-- __init__.py
```

You can download this example packages structure as a **bzip-file**. If we import the package "sound" by using the statement `import sound`, the package `sound` but not the subpackages `effects`, `filters` and `formats` will be imported, as we will see in the following example. The reason for this consists in the fact that the file `__init__.py` doesn't contain any code for importing subpackages:

```
>>> import sound
sound package is getting imported!
>>> sound
<module 'sound' from '/home/bernd/packages/sound/__init__.py'>
>>> sound.effects
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'sound' has no attribute 'effects'
```

If you also want to use the package `effects`, you have to import it explicitly with `import sound.effects`:

```
>>> import sound.effects
effects package is getting imported!
>>> sound.effects
<module 'sound.effects' from
'/home/bernd/packages/sound/effects/__init__.py'>
```

It is possible to have it done automatically when importing the `sound` module. For this purpose, we have to add the code line `import sound.effects` into the file `__init__.py` of the directory `sound`.

The file should look like this now:

```
"""An empty sound package

This is the sound package, providing hardly anything!

import sound.effects
print("sound package is getting imported!")
```

If we import the package `sound` from the interactive Python shell, we will see that the subpackage `effects` will also be automatically loaded:

```
>>> import sound
effects package is getting imported!
sound package is getting imported!
```

Instead of using an absolute path we could have imported the `effects`-package relative to the `sound` package:

```
"""An empty sound package

This is the sound package, providing hardly anything!

from . import effects
print("sound package is getting imported!")
```

It is also possible to automatically import the package `formats`, when we are importing the `effects` package. We can also do this with a relative path, which we will include into the `__init__.py` file of the directory `effects`:

```
from .. import formats
```

Importing `sound` will also automatically import the modules `formats` and `effects`:

```
>>> import sound
formats package is getting imported!
effects package is getting imported!
sound package is getting imported!
```

You can download the sound package structure with all the changes we have made so far as a **bzip-file**. To end this subchapter we want to show how to import the module `karaoke` from the package `filters` when we import the `effects` package. For this purpose we add the line `from ..filters import karaoke` into the `__init__.py` file of the directory `effects`. The complete file looks now like this:

```
"""An empty effects package

This is the effects package, providing hardly anything!"""

from .. import formats
from ..filters import karaoke
print("effects package is getting imported!")
```

Importing `sound` results in the following output:

```
>>> import sound
formats package is getting imported!
filters package is getting imported!
Module karaoke.py has been loaded!
effects package is getting imported!
sound package is getting imported!
```

We can access and use the functions of `karaoke` now:

```
>>> sound.filters.karaoke.func1()
Funktion func1 has been called!
>>>
```

IMPORTING A COMPLETE PACKAGE

For the next subchapter we will use again the initial example from the previous subchapter of our tutorial. We will add a module (file) `foobar` (filename: `foobar`) to the `sound` directory. The complete package can again be downloaded as a **bzip-file**. We want to demonstrate now, what happens, if we import the `sound` package with the star, i.e. `from sound import *`. Somebody might expect to import this way all the submodules and subpackages of the package. Let's see what happens:

```
>>> from sound import *
sound package is getting imported!
```

So we get the comforting message that the sound package has been imported. Yet, if we check with the `dir` function, we see that neither the module `foobar` nor the subpackages `effects`, `filters` and `formats` have been imported:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

Python provides a mechanism to give an explicit index of the subpackages and modules of a packages, which should be imported. For this purpose, we can define a list named `__all__`. This list will be taken as the list of module and package names to be imported when `from package import *` is encountered.

We will add now the line

```
__all__ = ["formats", "filters", "effects", "foobar"]
```

to the `__init__.py` file of the sound directory. We get a completely different result now:

```
>>> from sound import *
sound package is getting imported!
formats package is getting imported!
filters package is getting imported!
effects package is getting imported!
The module foobar is getting imported
>>>
```

Even though it is already apparent that all the modules have been imported, we can check with `dir` again:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'effects', 'filters', 'foobar', 'formats']
>>>
```

The next question is what will be imported, if we use `*` in a subpackage:

```
>>> from sound.effects import *
sound package is getting imported!
effects package is getting imported!
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
>>>
```

Like expected the modules inside of `effects` have not been imported automatically. So we can add the following `__all__` list into the `__init__` file of the package `effects`:

```
__all__ = ["echo", "surround", "reverse"]
```

Now we get the intended result:

```
>>> from sound.effects import *
sound package is getting imported!
```

```
effects package is getting imported!
Module echo.py has been loaded!
Module surround.py has been loaded!
Module reverse.py has been loaded!
>>>
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'echo', 'reverse', 'surround']
>>>
```

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice. The recommended way is to import specific modules from a package instead of using *.

REGULAR EXPRESSIONS

The aim of this chapter of our Python tutorial is to present a detailed and descriptive introduction into regular expressions. This introduction will explain the theoretical aspects of regular expressions and will show you how to use them in Python scripts.

The term "regular expression", sometimes also called regex or regexp, is originated in theoretical computer science. In theoretical computer science they are used to define a language family with certain characteristics, the so-called regular languages. A finite state machine (FSM), which accept language defined by a regular expression, exists for every regular expression. You can find an implementation of a ([Finite State Machine in Python](#)) on our website.

Regular Expressions are used in programming languages to filter texts or textstrings. It's possible to check, if a text or a string matches a regular expression. A great thing about regular expressions: The syntax of regular expressions is the same for all programming and script languages, e.g. Python, Perl, Java, SED, AWK and even X#.

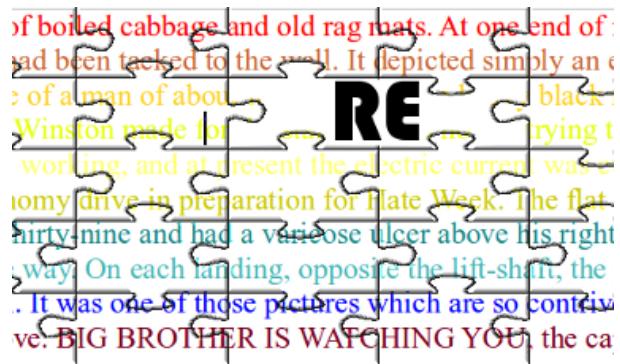
The first programs which had incorporated the capability to use regular expressions were the Unix tools ed (editor), the stream editor sed and the filter grep.

There is another mechanism in operating systems, which shouldn't be mistaken for regular expressions. Wildcards, also known as globbing, look very similar in their syntax to regular expressions. But the semantics differs considerably. Globbing is known from many command line shells, like the Bourne shell, the Bash shell or even DOS. In Bash e.g. the command "ls *.txt" lists all files (or even directories) ending with the suffix .txt; in regular expression notation "*.txt" wouldn't make sense, it would have to be written as ".*.txt"

INTRODUCTION

When we introduced the sequential data types, we got to know the "in" operator. We check in the following example, if the string "easily" is a substring of the string "Regular expressions easily explained!":

```
>>> s = "Regular expressions easily
explained!"
>>> "easily" in s
```



```
True
>>>
```

We show step by step with the following diagrams how this matching is performed:
We check if the string sub = "abc"

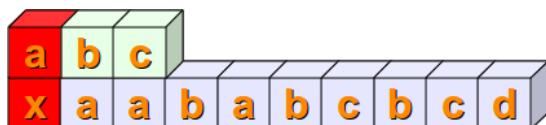


is contained in the string s = "xaababcbcd"

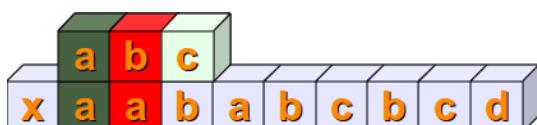


By the way, the string sub = "abc" can be seen as a regular expression, just a very simple one.

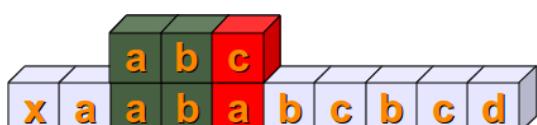
In the first place, we check, if the first positions of the two string match, i.e. $s[0] == \text{sub}[0]$. This is not satisfied in our example. We mark this fact by the colour red:



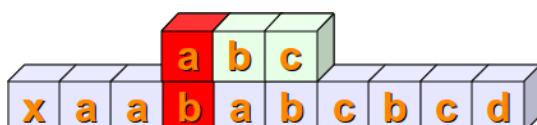
Then we check, if $s[1:4] == \text{sub}$. This means that we have to check at first, if $\text{sub}[0]$ is equal to $s[1]$. This is true and we mark it with the colour green. Then we have to compare the next positions. $s[2]$ is not equal to $\text{sub}[1]$, so we don't have to proceed further with the next position of sub and s:



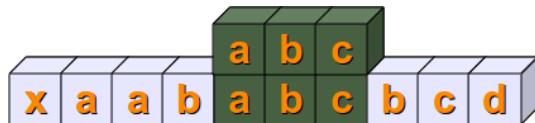
Now we have to check if $s[2:5]$ and sub are equal. The first two positions are equal but not the third:



The following steps should be clear without any explanations:



Finally, we have a complete match with `s[4:7] == sub` :



A SIMPLE REGULAR EXPRESSION

We already said in the previous section that we can see the variable "sub" from the introduction as a very simple regular expression.

If you want to use regular expressions in Python, you have to import the `re` module, which provides methods and functions to deal with regular expressions.

REPRESENTING REGULAR EXPRESSIONS IN PYTHON

From other languages you might be used to represent regular expressions within Slashes "/", e.g. that's the way Perl, SED or AWK deals with them. In Python there is no special notation. Regular expressions are represented as normal strings.

But this convenience brings along a small problem: The backslash is a special character used in regular expressions, but is also used as an escape character in strings. This implies that Python would first evaluate every backslash of a string and after this - without the necessary backslashes - it would be used as a regular expression. One way to prevent this consists in writing every backslash as "\\\" and this way keep it for the evaluation of the regular expression. This can give rise to extremely clumsy expressions. E.g. a backslash in a regular expression has to be written as a double backslash, because the backslash functions as an escape character in regular expressions. Therefore it has to be quoted. The same is true for Python strings. The backslash has to be quoted by a backslash. So, a regular expression to match the Windows path "C:\programs" corresponds to a string in regular expression notation with four backslashes, i.e. "C:\\\\programs".

The best way to overcome this problem consists in marking regular expressions as raw strings. The solution to our Windows path example looks as a raw string like this:

```
r"C:\\programs"
```

Let's look at another example, which might be quite disturbing for people used to wildcards:

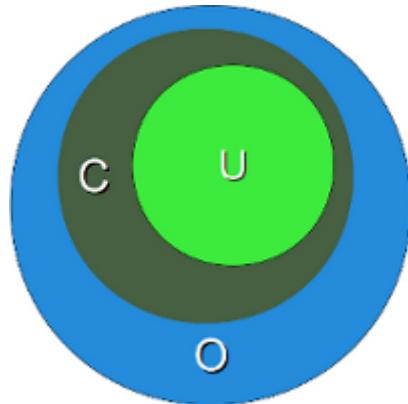
```
r"^.+\\.html$"
```

The regular expression of our previous example matches all file names (strings) which start with an "a" and end with ".html". We will explain in the following sections the structure of the example above in detail.

SYNTAX OF REGULAR EXPRESSIONS

`r"cat"` is a regular expression, though a very simple one without any metacharacters. Our RE `r"cat"` matches, for example, the following string: "A cat and a rat can't be friends."

Interestingly, the previous example shows already a "favourite" example for a mistake, frequently made not only by beginners and novices but also by advanced users of regular expressions. The idea of this example is to match strings containing the word "cat". We are successful with this, but unfortunately we are matching a lot of other words as well. If we match "cats" in a string that might be still okay, but what about all those words containing this character sequence "cat"? We match words like "education", "communicate", "falsification", "ramifications", "cattle" and many more. This is a case of "over matching", i.e. we receive positive results, which are wrong according to the problem we want to solve.



We have illustrated this problem in the diagram on the right side. The dark green Circle C corresponds to the set of "objects" we want to recognize. But instead we match all the elements of the set O (blue circle). C is a subset of O.

The set U (light green circle) in this diagram is a subset of C. U is a case of "under matching", i.e. if the regular expression is not matching all the intended strings. If we try to fix the previous RE, so that it doesn't create over matching, we might try the expression `r"cat "`. These blanks prevent the matching of the above mentioned words like "education", "falsification" and "ramification", but we fall prey to another mistake. What about the string "The cat, called Oscar, climbed on the roof."? The problem is that we don't expect a comma but only a blank behind the word "cat".

Before we go on with the description of the syntax of regular expressions, we want to explain how to use them in Python:

```
>>> import re
>>> x = re.search("cat", "A cat and a rat can't be friends.")
>>> print(x)
<_sre.SRE_Match object at 0x7fd4bf238238>
>>> x = re.search("cow", "A cat and a rat can't be friends.")
>>> print(x)
None
```

In the previous example we had to import the module `re` to be able to work with regular expressions. Then we used the method `search` from the `re` module. This is most probably the most important and the most often used method of this module. `re.search(expr,s)` checks a string `s` for an occurrence of a substring which matches the regular expression `expr`. The first substring (from left), which satisfies this condition will be returned. If a match has been possible, we get a so-called match object as a result, otherwise the value will be `None`. This method is already enough to use regular expressions in a basic way in Python programs. We can use it in conditional statements: If a regular expression matches, we get an SRE object returned, which is taken as a `True` value, and `None`, which is the return value if it doesn't match, is taken as `False`:

```
>>> if re.search("cat","A cat and a rat can't be friends."):
...     print("Some kind of cat has been found :-)")
... else:
...     print("No cat has been found :-)")
...
Some kind of cat has been found :-)
>>> if re.search("cow","A cat and a rat can't be friends."):
...     print("Cats and Rats and a cow.")
... else:
...     print("No cow around.")
...
No cow around.
```

ANY CHARACTER

Let's assume that we have not been interested in the previous example to recognize the word `cat`, but all three letter words, which end with "`at`".

The syntax of regular expressions supplies a metacharacter `.`, which is used like a placeholder for "any character". The regular expression of our example can be written like this:

```
r" .at "
```

This RE matches three letter words, isolated by blanks, which end in "`at`". Now we get words like "`rat`", "`cat`", "`bat`", "`eat`", "`sat`" and many others.

But what, if the text contains "words" like "`@at`" or "`3at`"? These words match as well and this means that we have created over matching again. We will learn a solution in the following section:

CHARACTER CLASSES

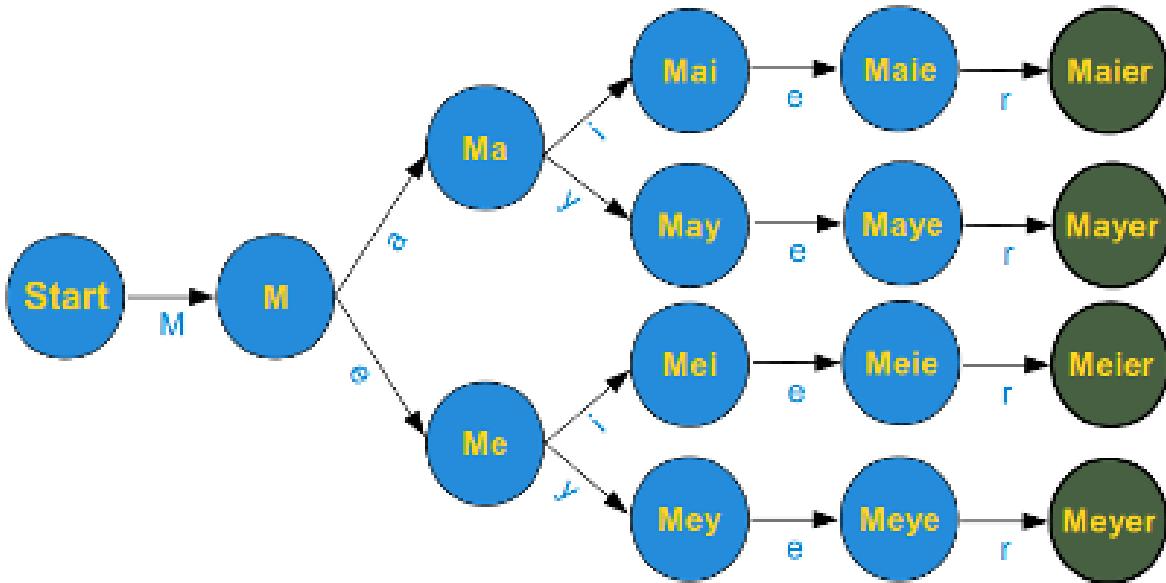
Square brackets, "[" and "]", are used to include a character class. `[xyz]` means e.g. either an "`x`", an "`y`" or a "`z`".

Let's look at a more practical example:

```
r"M[ae][iy]er"
```

This is a regular expression, which matches a surname which is quite common in German. A name with the same pronunciation and four different spellings: Maier, Mayer, Meier, Meyer

A finite state automata to recognize this expression can be build like this:



The graph of the finite state machine (FSM) is simplified to keep the design easy. There should be an arrow in the start node pointing back on its own, i.e. if a character other than an upper case "M" has been processed, the machine should stay in the start condition. Furthermore, there should be an arrow pointing back from all nodes except the final nodes (the green ones) to the start node, if not the expected letter has been processed. E.g. if the machine is in state Ma, after having processed an "M" and an "a", the machine has to go back to state "Start", if any character except "i" or "y" can be read. Those who have problems with this FSM, shouldn't be bothered, because it is not necessary to understand it for the things to come.

Instead of a choice between two characters, we often need a choice between larger character classes. We might need e.g. a class of letters between "a" and "e" or between "0" and "5"

To manage such character classes the syntax of regular expressions supplies a metacharacter "-". `[a-e]` a simplified writing for `[abcde]` or `[0-5]` denotes `[012345]`.

The advantage is obvious and even more impressive, if we have to coin expressions like "any uppercase letter" into regular expressions. So instead of `[ABCDEFGHIJKLMNPQRSTUVWXYZ]` we can write `[A-Z]`. If this is not convincing: Write an expression for the character class "any lower case or uppercase letter" `[A-Za-z]`

There is something more about the dash, we used to mark the begin and the end of a character class. The dash has only a special meaning if it is used within square brackets and in this case only if it isn't positioned directly after an opening or immediately in front of a closing bracket.

So the expression `[-az]` is only the choice between the three characters "-", "a" and "z", but no other characters. The same is true for `[az-]`.

Exercise:

What character class is described by `[-a-z]`?

Answer The character "-" and all the characters "a", "b", "c" all the way up to "z".

The only other special character inside square brackets (character class choice) is the caret "`^`". If it is used directly after an opening square bracket, it negates the choice. `[^0-9]` denotes the choice "any character but a digit". The position of the caret within the square brackets is crucial. If it is not positioned as the first character following the opening square bracket, it has no special meaning.

`[^abc]` means anything but an "a", "b" or "c"

`[a^bc]` means an "a", "b", "c" or a "`^`"

A PRACTICAL EXERCISE IN PYTHON

Before we go on with our introduction into regular expressions, we want to insert a practical exercise with Python.

We have a **phone list** of the Simpsons, yes, the famous Simpsons from the American animated TV series. There are some people with the surname Neu. We are looking for a Neu, but we don't know the first name, we just know that it starts with a J. Let's write a Python script, which finds all the lines of the phone book, which contain a person with the described surname and a first name starting with J. If you don't know how to read and work with files, you should work through our chapter File Management. So here is our example script:

```
import re

fh = open("simpsons_phone_book.txt")
for line in fh:
    if re.search(r"J.*Neu", line):
        print(line.rstrip())
fh.close()
```

The program above returns the following results:

```
Jack Neu 555-7666
Jeb Neu 555-5543
Jennifer Neu 555-3652
```

Instead of downloading `simpsons_phone_book.txt`, we can use the file directly from the website by using `urlopen` from the module `urllib.request`:

```

import re

from urllib.request import urlopen
with urlopen('https://www.python-course.eu/simpsons_phone_book.txt')
as fh:
    for line in fh:
        # line is a byte string so we transform it to utf-8:
        line = line.decode('utf-8').rstrip()
        if re.search(r"J.*Neu",line):
            print(line)

```

PREDEFINED CHARACTER CLASSES

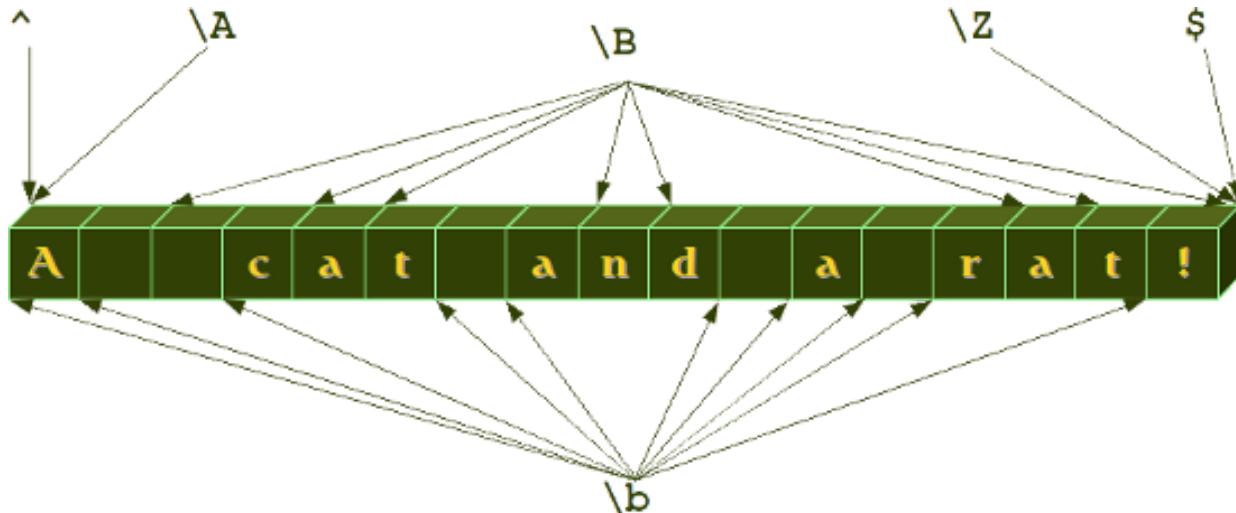
You might have realized that it can be quite cumbersome to construe certain character classes. A good example is the character class, which describes a valid word character. These are all lower case and uppercase characters plus all the digits and the underscore, corresponding to the following regular expression: `r"[a-zA-Z0-9_]"`

The special sequences consist of "\\" and a character from the following list:

- \d Matches any decimal digit; equivalent to the set [0-9].
- \D The complement of \d. It matches any non-digit character; equivalent to the set [^0-9].
- \s Matches any whitespace character; equivalent to [\t\n\r\f\v].
- \S The complement of \s. It matches any non-whitespace character; equiv. to [^\t\n\r\f\v].
- \w Matches any alphanumeric character; equivalent to [a-zA-Z0-9_]. With LOCALE, it will match the set [a-zA-Z0-9_] plus characters defined as letters for the current locale.
- \W Matches the complement of \w.
- \b Matches the empty string, but only at the start or end of a word.
- \B Matches the empty string, but not at the start or end of a word.
- \\" Matches a literal backslash.

WORD BOUNDARIES

The \b and \B of the previous overview of special sequences, is often not properly understood or even misunderstood especially by novices. While the other sequences match characters, - e.g. \w matches characters like "a", "b", "m", "3" and so on, - \b and \B don't match a character. They match empty strings depending on their neighbourhood, i.e. what kind of a character the predecessor and the successor is. So \b matches any empty string between a \W and a \w character and also between a \w and a \W character. \B is the complement, i.e empty strings between \W and \W or empty strings between \w and \w. We illustrate this in the following example:



We will get to know further "virtual" matching character, i.e. the caret (^), which is used to mark the beginning of a string, and the dollar sign (\$), which is used to mark the end of a string, respectively. \A and \Z, which can also be found in our previous diagram, are very seldom used alternatives to the caret and the dollar sign.

MATCHING BEGINNING AND END

As we have carried out previously in this introduction, the expression `r"m[ae][iy]er"` is capable of matching various spellings of the name Mayer and the name can be anywhere in the string:

```
>>> import re
>>> line = "He is a German called Mayer."
>>> if re.search(r"m[ae][iy]er", line): print("I found one!")
...
I found one!
>>>
```

But what, if we want to match a regular expression at the beginning of a string and only at the beginning?

The `re` module of Python provides two functions to match regular expressions. We have met already one of them, i.e. `search()`. The other has in our opinion a misleading name: `match()`

Misleading, because `match(re_str, s)` checks for a match of `re_str` merely at the beginning of the string.

But anyway, `match()` is the solution to our question, as we can see in the following example:

```
>>> import re
>>> s1 = "Mayer is a very common Name"
>>> s2 = "He is called Meyer but he isn't German."
>>> print(re.search(r"M[ae][iy]er", s1))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.search(r"M[ae][iy]er", s2))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.match(r"M[ae][iy]er", s1))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.match(r"M[ae][iy]er", s2))
None
>>>
```

So, this is a way to match the start of a string, but it's a Python specific method, i.e. it can't be used in other languages like Perl, AWK and so on. There is a general solution which is a standard for regular expressions:

The caret '^' matches the start of the string, and in MULTILINE (will be explained further down) mode also matches immediately after each newline, which the Python method `match()` doesn't do.

The caret has to be the first character of a regular expression:

```
>>> import re
>>> s1 = "Mayer is a very common Name"
>>> s2 = "He is called Meyer but he isn't German."
>>> print(re.search(r"^M[ae][iy]er", s1))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.search(r"^M[ae][iy]er", s2))
None
```

But what happens if we concatenate the two strings `s1` and `s2` in the following way:

```
s = s2 + "\n" + s1
```

Now the string doesn't start with a Maier of any kind, but the name is following a newline character:

```
>>> s = s2 + "\n" + s1
>>> print(re.search(r"^M[ae][iy]er", s))
None
>>>
```

The name hasn't been found, because only the beginning of the string is checked. It changes, if we use the multiline mode, which can be activated by adding the following third parameters to search:

```
>>> print(re.search(r"^M[ae][iy]er", s, re.MULTILINE))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.search(r"^M[ae][iy]er", s, re.M))
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.match(r"^M[ae][iy]er", s, re.M))
None
>>>
```

The previous example also shows that the multiline mode doesn't affect the match method. match() never checks anything but the beginning of the string for a match.

We have learnt how to match the beginning of a string. What about the end? Of course that's possible to. The dollar sign "*" is used as a metacharacter for this purpose."*" matches the end of a string or just before the newline at the end of the string. If in MULTILINE mode, it also matches before a newline. We demonstrate the usage of the "\$" character in the following example:

```
>>> print(re.search(r"Python\.$", "I like Python."))
<sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print(re.search(r"Python\.$", "I like Python and Perl."))
None
>>> print(re.search(r"Python\.$", "I like Python.\nSome prefer Java
or Perl."))
None
>>> print(re.search(r"Python\.$", "I like Python.\nSome prefer Java
or Perl.", re.M))
<sre.SRE_Match object at 0x7fc59c5f26b0>
>>>
```

OPTIONAL ITEMS

If you thought that our collection of Mayer names was complete, you were wrong. There are other ones all over the world, e.g. London and Paris, who dropped their "e". So we have four more names ["Mayr", "Meyr", "Meir", "Mair"] plus our old set ["Mayer", "Meyer", "Meier", "Maier"].

If we try to figure out a fitting regular expression, we realize that we miss something. A way to tell the computer "this "e" may or may not occur". A question mark is used as a notation for this. A question mark declares that the preceding character or expression is optional.

The final Mayer-Recognizer looks now like this:

```
r"M[ae][iy]e?r"
```

A subexpression is grouped by round brackets and a question mark following such a group means that this group may or may not exist. With the following expression we can match dates like "Feb 2011" or February 2011":

```
r"Feb(ruary)? 2011"
```

QUANTIFIERS

If you just use what we have introduced so far, you will still need a lot of things, above all some way of repeating characters or regular expressions. For this purpose, quantifiers are used. We have encountered one in the previous paragraph, i.e. the question mark.

A quantifier after a token, which can be a single character or group in brackets, specifies how often that preceding element is allowed to occur. The most common quantifiers are

- the question mark ?
- the asterisk or star character *, which is derived from the Kleene star
- and the plus sign +, derived from the Kleene cross

We have already used previously one of these quantifiers without explaining it, i.e. the asterisk. A star following a character or a subexpression group means that this expression or character may be repeated arbitrarily, even zero times.

```
r"[0-9]*"
```

The above expression matches any sequence of digits, even the empty string. `r".*"` matches any sequence of characters and the empty string.

Exercise:

Write a regular expression which matches strings which starts with a sequence of digits - at least one digit - followed by a blank.

Solution:

```
r"^[0-9][0-9]* "
```

So, you used the plus character "+". That's fine, but in this case you have either cheated by going ahead in the text or you know already more about regular expressions than we have covered in our course :-)

Now that we mentioned it: The plus operator is very convenient to solve the previous exercise. The plus operator is very similar to the star operator, except that the character or subexpression followed by a "+" sign has to be repeated at least one time. Here follows the solution to our exercise with the plus quantifier:

Solution with the plus quantifier:

```
r"^[0-9]+ "
```

If you work for a while with this arsenal of operators, you will miss inevitably at some point the possibility to repeat expressions for an exact number of times. Let's assume you

want to recognize the last lines of addresses on envelopes in Switzerland. These lines usually contain a four digits long post code followed by a blank and a city name. Using + or * are too unspecific for our purpose and the following expression seems to be too clumsy:

```
r"^[0-9][0-9][0-9][0-9] [A-Za-z]+"
```

Fortunately, there is an alternative available:

```
r"^[0-9]{4} [A-Za-z]*"
```

Now we want to improve our regular expression. Let's assume that there is no city name in Switzerland, which consists of less than 3 letters, at least 3 letters. We can denote this by [A-Za-z]{3,}. Now we have to recognize lines with German post code (5 digits) lines as well, i.e. the post code can now consist of either four or five digits:

```
r"^[0-9]{4,5} [A-Z][a-z]{2,}"
```

The general syntax is {from, to}: this means that the expression has to appear at least "from" times and not more than "to" times. {, to} is an abbreviated spelling for {0,to} and {from,} is an abbreviation for "at least from times but no upper limit"

GROUPING

We can group a part of a regular expression by surrounding it with parenthesis (round brackets). This way we can apply operators to the complete group instead of a single character.

CAPTURING GROUPS AND BACK REFERENCES

Parenthesis (round brackets, braces) not only group subexpressions but they create back references as well. The part of the string matched by the grouped part of the regular expression, i.e. the subexpression in parenthesis, is stored in a back reference. With the aid of back references we can reuse parts of regular expressions. These stored values can be both reused inside the expression itself and afterwards, when the regeexpr will have been executed. Before we continue with our treatise about back references, we want to strew in a paragraph about match objects, which is important for our next examples with back references.

A CLOSER LOOK AT THE MATCH OBJECTS

So far we have just checked, if an expression matched or not. We used the fact the re.search() returns a match object if it matches and None otherwise. We haven't been interested e.g. in what has been matched. The match object contains a lot of data about what has been matched, positions and so on.

A match object contains the methods group(), span(), start() and end(), as can be seen in the following application:

```
>>> import re
>>> mo = re.search("[0-9]+", "Customer number: 232454, Date:
February 12, 2011")
>>> mo.group()
'232454'
>>> mo.span()
(17, 23)
>>> mo.start()
17
>>> mo.end()
23
>>> mo.span()[0]
17
>>> mo.span()[1]
23
>>>
```

These methods are not difficult to understand. span() returns a tuple with the start and end position, i.e. the string index where the regular expression started matching in the string and ended matching. The methods start() and end() are in a way superfluous as the information is contained in span(), i.e. span()[0] is equal to start() and span()[1] is equal to end(). group(), if called without argument, returns the substring, which had been matched by the complete regular expression. With the help of group() we are also capable of accessing the matched substring by grouping parentheses, to get the matched substring of the n-th group, we call group() with the argument n: group(n).

We can also call group with more than integer argument, e.g. group(n,m). group(n,m) - provided there exists a subgoup n and m - returns a tuple with the matched substrings. group(n,m) is equal to (group(n), group(m)):

```
>>> import re
>>> mo = re.search("([0-9]+).*: (.*)", "Customer number: 232454,
Date: February 12, 2011")
>>> mo.group()
'232454, Date: February 12, 2011'
>>> mo.group(1)
'232454'
>>> mo.group(2)
'February 12, 2011'
>>> mo.group(1,2)
('232454', 'February 12, 2011')
>>>
```

A very intuitive example are XML or HTML tags. E.g. let's assume we have a file (called "tags.txt") with content like this:

```
<composer>Wolfgang Amadeus Mozart</composer>
<author>Samuel Beckett</author>
<city>London</city>
```

We want to rewrite this text automatically to

```
composer: Wolfgang Amadeus Mozart
author: Samuel Beckett
city: London
```

The following little Python script does the trick. The core of this script is the regular expression. This regular expression works like this: It tries to match a less than symbol "<". After this it is reading lower case letters until it reaches the greater than symbol. Everything encountered within "<" and ">" has been stored in a back reference which can be accessed within the expression by writing \1. Let's assume \1 contains the value "composer". When the expression has reached the first ">", it continues matching, as the original expression had been "<composer>(.*)</composer>":

```
import re
fh = open("tags.txt")
for i in fh:
    res = re.search(r"<([a-z]+)>(.*)</\1>", i)
    print(res.group(1) + ": " + res.group(2))
```

If there are more than one pair of parenthesis (round brackets) inside the expression, the backreferences are numbered \1, \2, \3, in the order of the pairs of parenthesis.

Exercise:

The next Python example makes use of three back references. We have an imaginary phone list of the Simpsons in a list. Not all entries contain a phone number, but if a phone number exists it is the first part of an entry. Then follows separated by a blank a surname, which is followed by first names. Surname and first name are separated by a comma. The task is to rewrite this example in the following way:

```
Allison Neu 555-8396
C. Montgomery Burns
Lionel Putz 555-5299
Homer Jay Simpson 555-7334
```

Python script solving the rearrangement problem:

```
import re

l = ["555-8396 Neu, Allison",
      "Burns, C. Montgomery",
      "555-5299 Putz, Lionel",
      "555-7334 Simpson, Homer Jay"]

for i in l:
    res = re.search(r"([0-9-]*) \s* ([A-Za-z]+), \s+ (.*)", i)
    print(res.group(3) + " " + res.group(2) + " " + res.group(1))
```

NAMED BACKREFERENCES

In the previous paragraph we introduced "Capturing Groups" and "Back references". More precisely, we could have called them "Numbered Capturing Groups" and "Numbered Back references".

Using capturing groups instead of "numbered" capturing groups allows you to assign descriptive names instead of automatic numbers to the groups. In the following example, we demonstrate this approach by catching the hours, minutes and seconds from a UNIX date string.

```
>>> import re
>>> s = "Sun Oct 14 13:47:03 CEST 2012"
>>> expr = r"\b(?P<hours>\d\d) : (?P<minutes>\d\d) : (?P<seconds>\d\d) \b"
>>> x = re.search(expr,s)
>>> x.group('hours')
'13'
>>> x.group('minutes')
'47'
>>> x.start('minutes')
14
>>> x.end('minutes')
16
>>> x.span('seconds')
(17, 19)
>>>
```

COMPREHENSIVE PYTHON EXERCISE

In this comprehensive exercise, we have to bring together the information of two files. In the first file, we have a list of nearly 15000 lines of post codes with the corresponding city names plus additional information. Here are some arbitrary lines of this file:

```
osm_id ort plz bundesland
1104550 Aach 78267 Baden-Württemberg
...
446465 Freiburg (Elbe) 21729 Niedersachsen
62768 Freiburg im Breisgau 79098 Baden-Württemberg
62768 Freiburg im Breisgau 79100 Baden-Württemberg
62768 Freiburg im Breisgau 79102 Baden-Württemberg
...
454863 Fulda 36037 Hessen
454863 Fulda 36039 Hessen
454863 Fulda 36041 Hessen
...
1451600 Gallin 19258 Mecklenburg-Vorpommern
449887 Gallin-Kuppentin 19386 Mecklenburg-Vorpommern
...
57082 Gärtringen 71116 Baden-Württemberg
1334113 Gartz (Oder) 16307 Brandenburg
...
2791802 Giengen an der Brenz 89522 Baden-Württemberg
```

```

2791802 Giengen an der Brenz 89537 Baden-Württemberg
...
1187159 Saarbrücken 66133 Saarland
1256034 Saarburg 54439 Rheinland-Pfalz
1184570 Saarlouis 66740 Saarland
1184566 Saarwellingen 66793 Saarland

```

The other file contains a list of the 19 largest German cities. Each line consists of the rank, the name of the city, the population, and the state (Bundesland):

1.	Berlin	3.382.169	Berlin
2.	Hamburg	1.715.392	Hamburg
3.	München	1.210.223	Bayern
4.	Köln	962.884	Nordrhein-Westfalen
5.	Frankfurt am Main	646.550	Hessen
6.	Essen	595.243	Nordrhein-Westfalen
7.	Dortmund	588.994	Nordrhein-Westfalen
8.	Stuttgart	583.874	Baden-Württemberg
9.	Düsseldorf	569.364	Nordrhein-Westfalen
10.	Bremen	539.403	Bremen
11.	Hannover	515.001	Niedersachsen
12.	Duisburg	514.915	Nordrhein-Westfalen
13.	Leipzig	493.208	Sachsen
14.	Nürnberg	488.400	Bayern
15.	Dresden	477.807	Sachsen
16.	Bochum	391.147	Nordrhein-Westfalen
17.	Wuppertal	366.434	Nordrhein-Westfalen
18.	Bielefeld	321.758	Nordrhein-Westfalen
19.	Mannheim	306.729	Baden-Württemberg

Our task is to create a list with the top 19 cities, with the city names accompanied by the postal code. If you want to test the following program, you have to save the list above in a file called `largest_cities_germany.txt` and you have to download and save the [list of German post codes](#)

```

import re

with open("zuordnung_plz_ort.txt", encoding="utf-8") as
fh_post_codes:
    codes4city = {}
    for line in fh_post_codes:
        res = re.search(r"[\d]+([^\d]+[a-z])\s+(\d+)", line)
        if res:
            city, post_code = res.groups()
            if city in codes4city:
                codes4city[city].add(post_code)
            else:
                codes4city[city] = {post_code}

with open("largest_cities_germany.txt", encoding="utf-8") as
fh_largest_cities:
    for line in fh_largest_cities:
        re_obj = re.search(r"^[0-9]{1,2}\.\s+([\w\s-]+\w)\s+[0-9]", line)
        city = re_obj.group(1)
        print(city, codes4city[city])

```

The output of this file looks like this, but we have left out all but the first three postal codes for every city:

```
Berlin {'10715', '13158', '13187', ...}
Hamburg {'22143', '22119', '22523', ...}
München {'80802', '80331', '80807', ...}
Köln {'51065', '50997', '51067', ...}
Frankfurt am Main {'65934', '60529', '60308', ...}
Essen {'45144', '45134', '45309', ... }
Dortmund {'44328', '44263', '44369',...}
Stuttgart {'70174', '70565', '70173', ...}
Düsseldorf {'40217', '40589', '40472', ...}
Bremen {'28207', '28717', '28777', ...}
Hannover {'30169', '30419', '30451', ...}
Duisburg {'47137', '47059', '47228', ...}
Leipzig {'4158', '4329', '4349', ...'}
Nürnberg {'90419', '90451', '90482', ...}
Dresden {'1217', '1169', '1324', ...}
Bochum {'44801', '44892', '44805', ...}
Wuppertal {'42109', '42119', '42287', ...}
Bielefeld {'33613', '33607', '33699', ...}
Mannheim {'68161', '68169', '68167', ...}
```

ANOTHER COMPREHENSIVE EXAMPLE

We want to present another real life example in our Python course. A regular expression for UK postcodes.

We write an expression, which is capable of recognizing the postal codes or postcodes of the UK.

Postcode units consist of between five and seven characters, which are separated into two parts by a space. The two to four characters before the space represent the so-called outward code or out code intended to direct mail from the sorting office to the delivery office. The part following the space, which consists of a digit followed by two uppercase characters, comprises the so-called inward code, which is needed to sort mail at the final delivery office. The last two uppercase characters do not use the letters CIKMOV, so as not to resemble digits or each other when hand-written.

The outward code can have the form: One or two uppercase characters, followed by either a digit or the letter R, optionally followed by an uppercase character or a digit. (We do not consider all the detailed rules for postcodes, i.e. only certain character sets are valid depending on the position and the context.)

A regular expression for matching this superset of UK postcodes looks like this:

```
r"\b[A-Z]{1,2}[0-9R]? [0-9A-Z]? [0-9] [ABD-HJLNP-UW-Z]{2}\b"
```

The following Python program uses the regexp above:

```
import re

example_codes = ["SW1A 0AA", # House of Commons
                 "SW1A 1AA", # Buckingham Palace
                 "SW1A 2AA", # Downing Street
                 "BX3 2BB", # Barclays Bank
                 "DH98 1BT", # British Telecom
                 "N1 9GU", # Guardian Newspaper
                 "E98 1TT", # The Times
                 "TIM E22", # a fake postcode
                 "A B1 A22", # not a valid postcode
                 "EC2N 2DB", # Deutsche Bank
                 "SE9 2UG", # University of Greenwich
                 "N1 0UY", # Islington, London
                 "EC1V 8DS", # Clerkenwell, London
                 "WC1X 9DT", # WC1X 9DT
                 "B42 1LG", # Birmingham
                 "B28 9AD", # Birmingham
                 "W12 7RJ", # London, BBC News Centre
                 "BBC 007" # a fake postcode
                ]

pc_re = r"[A-z]{1,2}[0-9R][0-9A-Z]?\ [0-9][ABD-HJLNP-UW-Z]{2}"

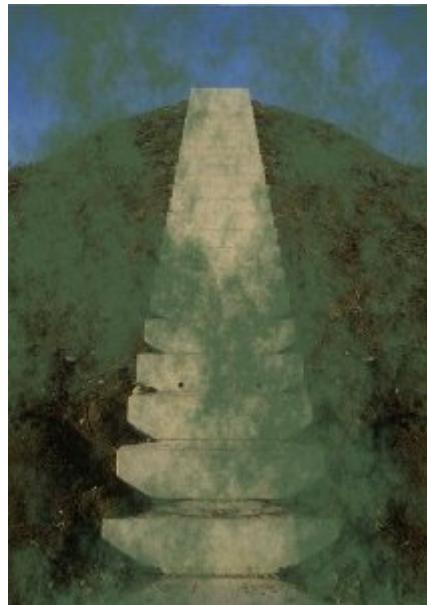
for postcode in example_codes:
    r = re.search(pc_re, postcode)
    if r:
        print(postcode + " matched!")
    else:
        print(postcode + " is not a valid postcode!")
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

ADVANCED REGULAR EXPRESSIONS

INTRODUCTION

In our [introduction to regular expressions](#) of our tutorial we have covered the basic principles of regular expressions. We have shown, what the simplest regular expression looks like. We have also learnt, how to use regular expressions in Python by using the search() and the match() methods of the re module. The concept of formulating and using character classes should be well known by now, as well as the predefined character classes like \d, \D, \s, \S, and so on. You should have learnt how to match the beginning and the end of a string with a regular expression. You should know the special meaning of the question mark to make items optional. We have also introduced the quantifiers to repeat characters and groups arbitrarily or in certain ranges.



You should also be familiar with the use of grouping and the syntax and usage of back references.

Furthermore, we had explained the match objects of the re module and the information they contain and how to retrieve this information by using the methods span(), start(), end(), and group().

The introduction ended with a comprehensive example in Python.

In this chapter we will continue with our explanations of the syntax of the regular expressions. We will also explain further methods of the Python module re. E.g. how to find all the matched substrings of a regular expression. A task which needs programming in other programming languages like Perl or Java, but can be dealt with the call of one method of the re module of Python. So far, we only know how to define a choice of characters with a character class. We will demonstrate in this chapter of our tutorial, how to formulate alternations of substrings.

FINDING ALL MATCHED SUBSTRINGS

The Python module `re` provides another great method, which other languages like Perl and Java don't provide. If you want to find all the substrings in a string, which match a regular expression, you have to use a loop in Perl and other languages, as can be seen in the following Perl snippet:

```
while ($string =~ m/regex/g) {
    print "Found '$&'. Next attempt at character " . pos($string)+1 .
"\n";
}
```

It's a lot easier in Python. No need to loop. We can just use the `findall` method of the `re` module:

```
re.findall(pattern, string[, flags])
```

`findall` returns all non-overlapping matches of `pattern` in `string`, as a list of strings. The `string` is scanned left-to-right, and matches are returned in the order in which they are found.

```
>>> t="A fat cat doesn't eat oat but a rat eats bats."
>>> mo = re.findall("[force]at", t)
>>> print(mo)
['fat', 'cat', 'eat', 'oat', 'rat', 'eat']
```

If one or more groups are present in the pattern, `findall` returns a list of groups. This will be a list of tuples if the pattern has more than one group. We demonstrate this in our next example. We have a long string with various Python training courses and their dates. With the first call to `findall`, we don't use any grouping and receive the complete string as a result. In the next call, we use grouping and `findall` returns a list of 2-tuples, each having the course name as the first component and the dates as the second component:

```
>>> import re
>>> courses = "Python Training Course for Beginners: 15/Aug/2011 - 19/Aug/2011; Python Training Course Intermediate: 12/Dec/2011 - 16/Dec/2011; Python Text Processing Course: 31/Oct/2011 - 4/Nov/2011"
>>> items = re.findall("[^:]*:[^;]*;?", courses)
>>> items
['Python Training Course for Beginners: 15/Aug/2011 - 19/Aug/2011;', 'Python Training Course Intermediate: 12/Dec/2011 - 16/Dec/2011;', 'Python Text Processing Course: 31/Oct/2011 - 4/Nov/2011']
>>> items = re.findall("([:^]*):([^;]*;?)", courses)
>>> items
[('Python Training Course for Beginners', '15/Aug/2011 - 19/Aug/2011;'), ('Python Training Course Intermediate', '12/Dec/2011 - 16/Dec/2011;'), ('Python Text Processing Course', '31/Oct/2011 - 4/Nov/2011')]
>>>
```

ALTERNATIONS

In our introduction to regular expressions we had introduced character classes. Character classes offer a choice out of a set of characters. Sometimes we need a choice between several regular expression. It's a logical "or" and that's why the symbol for this construct is the "|" symbol.

In the following example, we check, if one of the cities London, Paris, Zurich, Konstanz Bern or Strasbourg appear in a string preceded by the word "location":

```
>>> import re
>>> str = "Course location is London or Paris!"
>>> mo = re.search(r"location.*"
(London|Paris|Zurich|Strasbourg)",str)
>>> if mo: print(mo.group())
...
location is London or Paris
>>>
```

If you consider the previous example as too artificial, here is another one. Let's assume, you want to filter your email. You want to find all the correspondence (conversations) between you and Guido van Rossum, the creator and designer of Python. The following regular expression is helpful for this purpose:

```
r"(^To:|From:) (Guido|van Rossum)"
```

This expression matches all lines starting with either 'To:' or 'From:', followed by a space and then either by the first name 'Guido' or the surname 'van Rossum'.

COMPILING REGULAR EXPRESSIONS

If you want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.

The general syntax:

```
re.compile(pattern[, flags])
```

compile returns a regex object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

Abbreviation	Full name	Description
re.I	re.IGNORECASE	Makes the regular expression case-insensitive
re.L	re.LOCALE	The behaviour of some special sequences like \w, \W, \b, \s, \S will be made dependant on the current locale, i.e. the user's language, country aso.
re.M	re.MULTILINE	^ and \$ will match at the beginning and at the end of each line and not just at the beginning and the end of

		the string
re.S	re.DOTALL	The dot "." will match every character plus the newline
re.U	re.UNICODE	Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties
re.X	re.VERBOSE	Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class. # are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character can be used to start a comment.

Compiled regular objects usually are not saving much time, because Python internally compiles AND CACHES regexes whenever you use them with `re.search()` or `re.match()`. The only extra time a non-compiled regex takes is the time it needs to check the cache, which is a key lookup of a dictionary.

A good reason to use them is to separate the definition of a regex from its use.

EXAMPLE

We have already introduced a regular expression for matching a superset of UK postcodes in our introductory chapter:

```
r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
```

We demonstrate with this regular expression, how we can use the `compile` functionality of the module `re` in the following interactive session. The regular expression "regex" is compiled with `re.compile(regex)` and the compiled object is saved in the object `compiled_re`. Now we call the method `search()` of the object `compiled_re`:

```
>>> import re
>>> regex = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
>>> address = "BBC News Centre, London, W12 7RJ"
>>> compiled_re = re.compile(regex)
>>> res = compiled_re.search(address)
>>> print(res)
```

```
<_sre.SRE_Match object at 0x174e578>
>>>
```

SPLITTING A STRING WITH OR WITHOUT REGULAR EXPRESSIONS

There is a string method `split`, which can be used to split a string into a list of substrings.

```
str.split([sep[, maxsplit]])
```

As you can see, the method `split` has two optional parameters. If none is given (or is `None`) , a string will be separated into substring using whitespaces as delimiters, i.e. every substring consisting purely of whitespaces is used as a delimiter.



We demonstrate this behaviour with a famous quotation by Abraham Lincoln:

```
>>> law_courses = "Let reverence for the laws be breathed by every
American mother to the lisping babe that prattles on her lap. Let it
be taught in schools, in seminaries, and in colleges. Let it be
written in primers, spelling books, and in almanacs. Let it be
preached from the pulpit, proclaimed in legislative halls, and
```

```

enforced in the courts of justice. And, in short, let it become the
political religion of the nation."
>>> law_courses.split()
['Let', 'reverence', 'for', 'the', 'laws', 'be', 'breathed', 'by',
'every', 'American', 'mother', 'to', 'the', 'lisping', 'babe',
'that', 'prattles', 'on', 'her', 'lap.', 'Let', 'it', 'be',
'taught', 'in', 'schools', 'in', 'seminaries', 'and', 'in',
'colleges.', 'Let', 'it', 'be', 'written', 'in', 'primers',
'spelling', 'books', 'and', 'in', 'almanacs.', 'Let', 'it', 'be',
'preached', 'from', 'the', 'pulpit', 'proclaimed', 'in',
'legislative', 'halls', 'and', 'enforced', 'in', 'the', 'courts',
'of', 'justice.', 'And', 'in', 'short', 'let', 'it', 'become',
'the', 'political', 'religion', 'of', 'the', 'nation.']
>>>

```

Now we look at a string, which could stem from an Excel or an OpenOffice calc file. We have seen in our previous example that split takes whitespaces as default separators. We want to split the string in the following little example using semicolons as separators. The only thing we have to do is to use ";" as an argument of split():

```

>>> line = "James;Miller;teacher;Perl"
>>> line.split(";")
['James', 'Miller', 'teacher', 'Perl']

```

The method split() has another optional parameter: maxsplit.

If maxsplit is given, at most maxsplit splits are done. This means that the resulting list will have at most "maxsplit + 1" elements.

We will illustrate the mode of operation of maxsplit in the next example:

```

>>> mammon = "The god of the world's leading religion. The chief
temple is in the holy city of New York."
>>> mammon.split(" ",3)
['The', 'god', 'of', "the world's leading religion. The chief temple
is in the holy city of New York."]

```

We used a Blank as a delimiter string in the previous example, which can be a problem: If multiple blanks or whitespaces are connected, split() will split the string after every single blank, so that we will get empty strings and strings with only a tab inside ('\t') in our result list:

```

>>> mammon = "The god \t of the world's leading religion. The chief
temple is in the holy city of New York."
>>> mammon.split(" ",5)
['The', 'god', '', '\t', 'of', "the world's leading religion. The
chief temple is in the holy city of New York."]
>>>

```

We can prevent the separation of empty strings by using None as the first argument. Now split will use the default behaviour, i.e. every substring consisting of connected whitespace characters will be taken as one separator:

```
>>> mammon.split(None,5)
['The', 'god', 'of', 'the', "world's", 'leading religion. The chief
temple is in the holy city of New York.']}
```

REGULAR EXPRESSION SPLIT

The string method `split()` is the right tool in many cases, but what, if you want e.g. to get the bare words of a text, i.e. without any special characters and whitespaces. If we want this, we have to use the `split` function from the `re` module. We illustrate this method with a short text from the beginning of Metamorphoses by Ovid:

```
>>> import re
>>> metamorphoses = "OF bodies chang'd to various forms, I sing: Ye
Gods, from whom these miracles did spring, Inspire my numbers with
coelestial heat;"
>>> re.split("\W+",metamorphoses)
['OF', 'bodies', 'chang', 'd', 'to', 'various', 'forms', 'I',
'sing', 'Ye', 'Gods', 'from', 'whom', 'these', 'miracles', 'did',
'spring', 'Inspire', 'my', 'numbers', 'with', 'coelestial', 'heat',
'']
```

The following example is a good case, where the regular expression is really superior to the string `split`. Let's assume that we have data lines with surnames, first names and professions of names. We want to clear the data line of the superfluous and redundant text descriptions, i.e. "surname: ", "prename: " and so on, so that we have solely the surname in the first column, the first name in the second column and the profession in the third column:

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession:
president", "surname: Merkel, prename: Angela, profession:
chancellor"]
>>> for line in lines:
...     re.split(", * *\w*: ", line)
...
[ '', 'Obama', 'Barack', 'president']
[ '', 'Merkel', 'Angela', 'chancellor']
>>>
```

We can easily improve the script by using a slice operator, so that we don't have the empty string as the first element of our result lists:

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession:
president", "surname: Merkel, prename: Angela, profession:
chancellor"]
>>> for line in lines:
...     re.split(", * *\w*: ", line)[1:]
...
['Obama', 'Barack', 'president']
['Merkel', 'Angela', 'chancellor']
>>>
```

And now for something completely different: There is a connection between Barack Obama and Python, or better Monty Python. John Cleese, one of the members of Monty Python told the Western Daily Press in April 2008: "I'm going to offer my services to him as a speech writer because I think he is a brilliant man"

SEARCH AND REPLACE WITH SUB

```
re.sub(regex, replacement, subject)
```

Every match of the regular expression regex in the string subject will be replaced by the string replacement.

Example:

```
>>> import re
>>> str = "yes I said yes I will Yes."
>>> res = re.sub("[yY]es","no", str)
>>> print(res)
no I said no I will no.
```

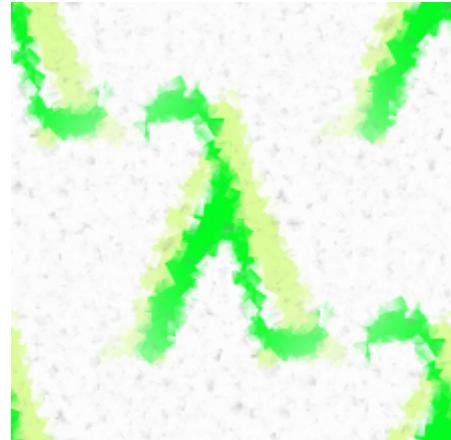

LAMBDA, FILTER, REDUCE AND MAP

LAMBDA OPERATOR

If Guido van Rossum, the author of the programming language Python, had got his will, this chapter would be missing in our tutorial. In his article from May 2005 "All Things Pythonic: The fate of reduce() in Python 3000", he gives his reasons for dropping lambda, map(), filter() and reduce(). He expected resistance from the Lisp and the scheme "folks". What he didn't anticipate was the rigidity of this opposition. Enough that Guido van Rossum wrote hardly a year later:

"After so many attempts to come up with an alternative for lambda, perhaps we should admit defeat. I've not had the time to follow the most recent rounds, but I propose that we keep lambda, so as to stop wasting everybody's talent and time on an impossible quest."

We can see the result: lambda, map() and filter() are still part of core Python. Only reduce() had to go; it moved into the module `functools`.



His reasoning for dropping them is like this:

- There is an equally powerful alternative to lambda, filter, map and reduce, i.e. [list comprehension](#)
- List comprehension is more evident and easier to understand
- Having both list comprehension and "Filter, map, reduce and lambda" is transgressing the Python motto "There should be one obvious way to solve a problem"

Some like it, others hate it and many are afraid of the lambda operator. The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers.

The general syntax of a lambda function is quite simple:

```
lambda argument_list: expression
```

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.

The following example of a lambda function returns the sum of its two arguments:

```
>>> sum = lambda x, y : x + y
>>> sum(3,4)
7
>>>
```

The above example might look like a plaything for a mathematician. A formalism which turns an easy to comprehend issue into an abstract harder to grasp formalism. Above all, we could have had the same effect by just using the following conventional function definition:

```
>>> def sum(x,y):
...     return x + y
...
>>> sum(3,4)
7
>>>
```

We can assure you that the advantages of this approach will be apparent, when you will have learnt to use the `map()` function.

THE MAP() FUNCTION

As we have mentioned earlier, the advantage of the lambda operator can be seen when it is used in combination with the `map()` function.

`map()` is a function which takes two arguments:

```
r = map(func, seq)
```

The first argument `func` is the name of a function and the second a sequence (e.g. a list) `seq`. `map()` applies the function `func` to all the elements of the sequence `seq`. Before Python3, `map()` used to return a list, where each element of the result list was the result of the function `func` applied on the corresponding element of the list or tuple "seq". With Python 3, `map()` returns an iterator.

The following example illustrates the way of working of `map()`:

```
>>> def fahrenheit(T):
...     return ((float(9)/5)*T + 32)
... 
```

```
>>> def celsius(T):
...     return (float(5)/9)*(T-32)
...
>>> temperatures = (36.5, 37, 37.5, 38, 39)
>>> F = map(fahrenheit, temperatures)
>>> C = map(celsius, F)
>>>
>>> temperatures_in_Fahrenheit = list(map(fahrenheit, temperatures))
>>> temperatures_in_Celsius = list(map(celsius,
temperatures_in_Fahrenheit))
>>> print(temperatures_in_Fahrenheit)
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
>>> print(temperatures_in_Celsius)
[36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]
>>>
```

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

```
>>> C = [39.2, 36.5, 37.3, 38, 37.8]
>>> F = list(map(lambda x: (float(9)/5)*x + 32, C))
>>> print(F)
[102.56, 97.7, 99.14, 100.4, 100.0399999999999]
>>> C = list(map(lambda x: (float(5)/9)*(x-32), F))
>>> print(C)
[39.2, 36.5, 37.30000000000004, 38.00000000000001, 37.8]
>>>
```

map() can be applied to more than one list. The lists don't have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

```
>>> a = [1, 2, 3, 4]
>>> b = [17, 12, 11, 10]
>>> c = [-1, -4, 5, 9]
>>> list(map(lambda x, y : x+y, a, b))
[18, 14, 14, 14]
>>> list(map(lambda x, y, z : x+y+z, a, b, c))
[17, 10, 19, 23]
>>> list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
[37.5, 33.0, 24.5, 21.0]
>>>
```

If one list has fewer elements than the others, map will stop when the shortest list has been consumed:

```
>>> a = [1, 2, 3]
>>> b = [17, 12, 11, 10]
>>> c = [-1, -4, 5, 9]
>>>
>>> list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
[37.5, 33.0, 24.5]
>>>
```

We can see in the example above that the parameter x gets its values from the list a, while y gets its values from b and z from list c.

MAPPING A LIST OF FUNCTIONS

The map function of the previous chapter was used to apply one function to one or more iterables. We will now write a function which applies a bunch of functions, which may be an iterable such as a list or a tuple, for example, to one Python object.

```
from math import sin, cos, tan, pi

def map_functions(x, functions):
    """ map an iterable of functions on the the object x """
    res = []
    for func in functions:
        res.append(func(x))
    return res

family_of_functions = (sin, cos, tan)
print(map_functions(pi, family_of_functions))
```

The previous program returns the following output:

```
[1.2246467991473532e-16, -1.0, -1.2246467991473532e-16]
```

The previously defined map_functions function can be simplified with the list comprehension technique, which we will cover in the chapter [list comprehension](#):

```
def map_functions(x, functions):
    return [ func(x) for func in functions ]
```

FILTERING

The function

```
filter(function, sequence)
```

offers an elegant way to filter out all the elements of a sequence "sequence", for which the function *function* returns True. i.e. an item will be produced by the iterator result of filter(function, sequence) if item is included in the sequence "sequence" and if function(item) returns True.

In other words: The function filter(f,l) needs a function f as its first argument. f has to

return a Boolean value, i.e. either True or False. This function will be applied to every element of the list l . Only if f returns True will the element be produced by the iterator, which is the return value of $\text{filter}(\text{function}, \text{sequence})$.

In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))
>>> print(odd_numbers)
[1, 1, 3, 5, 13, 21, 55]
>>> even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]
>>>
>>>
>>> # or alternatively:
...
>>> even_numbers = list(filter(lambda x: x % 2 -1, fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]
>>>
```

REDUCING A LIST

As we mentioned in the introduction of this chapter of our tutorial, $\text{reduce}()$ had been dropped from the core of Python when migrating to Python 3. Guido van Rossum hates $\text{reduce}()$, as we can learn from his statement in a posting, March 10, 2005, in artima.com:

*"So now $\text{reduce}()$. This is actually the one I've always hated most, because, apart from a few examples involving + or *, almost every time I see a $\text{reduce}()$ call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the $\text{reduce}()$ is supposed to do. So in my mind, the applicability of $\text{reduce}()$ is pretty much limited to associative operators, and in all other cases it's better to write out the accumulation loop explicitly."*

The function

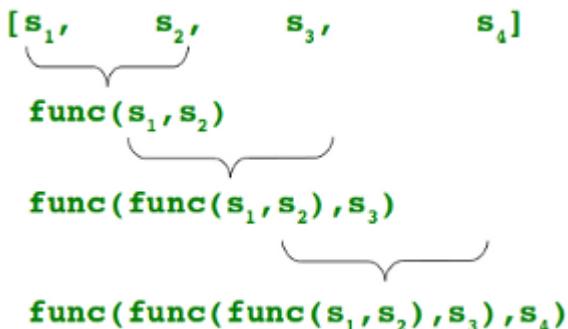
```
reduce(func, seq)
```

continually applies the function $\text{func}()$ to the sequence seq . It returns a single value.

If $\text{seq} = [s_1, s_2, s_3, \dots, s_n]$, calling $\text{reduce}(\text{func}, \text{seq})$ works like this:

- At first the first two elements of seq will be applied to func, i.e. $\text{func}(s_1, s_2)$ The list on which reduce() works looks now like this: [$\text{func}(s_1, s_2), s_3, \dots, s_n$]
- In the next step func will be applied on the previous result and the third element of the list, i.e. $\text{func}(\text{func}(s_1, s_2), s_3)$
The list looks like this now: [$\text{func}(\text{func}(s_1, s_2), s_3), \dots, s_n$]
- Continue like this until just one element is left and return this element as the result of reduce()

If n is equal to 4 the previous explanation can be illustrated like this:

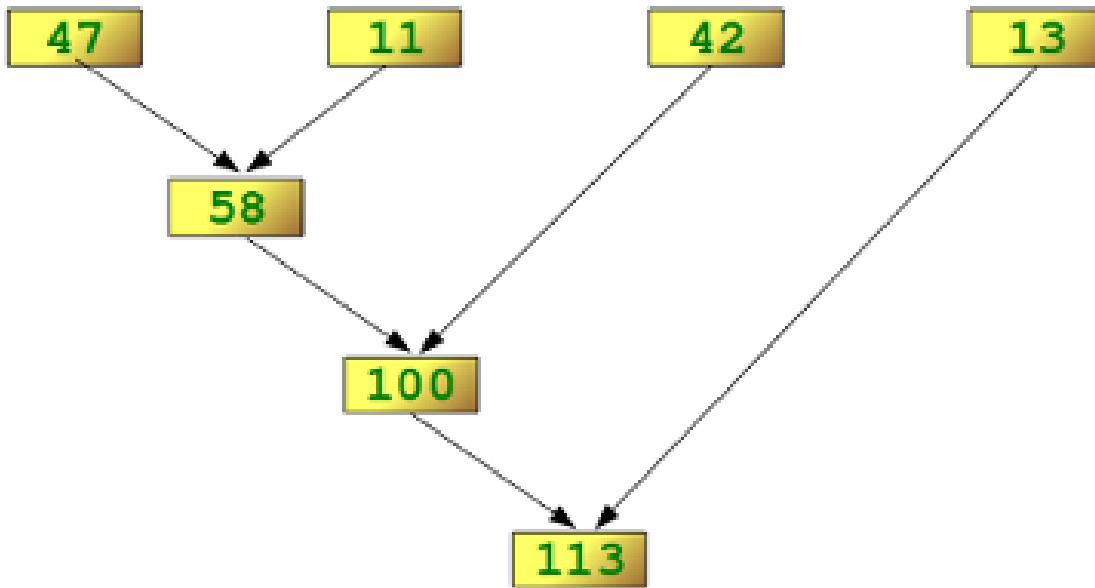


We want to illustrate this way of working of reduce() with a simple example. We have to import functools to be capable of using reduce:

```

>>> import functools
>>> functools.reduce(lambda x,y: x+y, [47,11,42,13])
113
>>>
  
```

The following diagram shows the intermediate steps of the calculation:



EXAMPLES OF REDUCE()

Determining the maximum of a list of numerical values by using reduce:

```

>>> from functools import reduce
>>> f = lambda a,b: a if (a > b) else b
>>> reduce(f, [47,11,42,102,13])
102
>>>
  
```

Calculating the sum of the numbers from 1 to 100:

```

>>> from functools import reduce
>>> reduce(lambda x, y: x+y, range(1,101))
5050
  
```

It's very simple to change the previous example to calculate the product (the factorial) from 1 to a number, but do not choose 100. We just have to turn the "+" operator into "*":

```

>>> reduce(lambda x, y: x*y, range(1,49))
124139155925360726708622890473733750385214863546777600000000000
  
```

If you are into lottery, here are the chances to win a 6 out of 49 drawing:

```

>>> reduce(lambda x, y: x*y, range(44,50)) / reduce(lambda x, y: x*y,
range(1,7))
13983816.0
>>>
  
```

EXERCISES

1. Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Order Number	Book Title and Author	Quantity	Price per Item
34587	Learning Python, Mark Lutz	4	40.95
98762	Programming Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95
88112	Einführung in Python3, Bernd Klein	3	24.99

Write a Python program, which returns a list with 2-tuples. Each tuple consists of a the order number and the product of the price per items and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.

Write a Python program using lambda and map.

2. The same bookshop, but this time we work on a different list. The sublists of our lists look like this:

[ordernumber, (article number, quantity, price per unit), ... (article number, quantity, price per unit)]

Write a program which returns a list of two tuples with (order number, total amount of order).

SOLUTIONS TO THE EXERCISES

```
1. orders = [ ["34587", "Learning Python, Mark Lutz", 4, 40.95],
              ["98762", "Programming Python, Mark Lutz", 5,
               56.80],
              ["77226", "Head First Python, Paul Barry", 3, 32.95],
              ["88112", "Einführung in Python3, Bernd Klein",
               3, 24.99]]

min_order = 100
invoice_totals = list(map(lambda x: x if x[1] >= min_order else
                           (x[0], x[1] + 10),
                           map(lambda x: (x[0], x[2] *
                                         x[3]), orders)))
```

```
print(invoice_totals)
```

The output of the previous program looks like this:

```
[('34587', 163.8), ('98762', 284.0), ('77226',
108.85000000000001), ('88112', 84.97)]
```

2. from functools import reduce

```
orders = [ [1, ("5464", 4, 9.99), ("8274", 18, 12.99), ("9744", 9,
44.95)],
           [2, ("5464", 9, 9.99), ("9744", 9, 44.95)],
           [3, ("5464", 9, 9.99), ("88112", 11, 24.99)],
           [4, ("8732", 7, 11.99), ("7733", 11, 18.99), ("88112",
5, 39.95)] ]

min_order = 100
invoice_totals = list(map(lambda x: [x[0]] + list(map(lambda y:
y[1]*y[2], x[1:])), orders))
invoice_totals = list(map(lambda x: [x[0]] + [reduce(lambda a,b:
a + b, x[1:])], invoice_totals))
invoice_totals = list(map(lambda x: x if x[1] >= min_order else
(x[0], x[1] + 10), invoice_totals))

print (invoice_totals)
```

We will get the following result:

```
[[1, 678.329999999999], [2, 494.4600000000004], [3,
364.7999999999995], [4, 492.57]]
```

LIST COMPREHENSION

INTRODUCTION

We learned in the previous chapter "[Lambda Operator, Filter, Reduce and Map](#)" that Guido van Rossum prefers list comprehensions to constructs using map, filter, reduce and lambda. In this chapter we will cover the essentials about list comprehensions.



List comprehensions were added with Python 2.0. Essentially, it is Python's way of implementing a well-known notation for sets as used by mathematicians. In mathematics the square numbers of the natural numbers are, for example, created by $\{ x^2 \mid x \in \mathbb{N} \}$ or the set of complex integers $\{ (x,y) \mid x \in \mathbb{Z} \wedge y \in \mathbb{Z} \}$.

List comprehension is an elegant way to define and create list in Python. These lists have often the qualities of sets, but are not in all cases sets.

List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce(). For most people the syntax of list comprehension is easier to be grasped.

EXAMPLES

In the chapter on lambda and map() we had designed a map() function to convert Celsius values into Fahrenheit and vice versa. It looks like this with list comprehension:

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
>>> print(Fahrenheit)
[102.56, 97.70000000000003, 99.14000000000001, 100.0399999999999]
```

A Pythagorean triple consists of three positive integers a , b , and c , such that $a^2 + b^2 = c^2$.

Such a triple is commonly written (a, b, c) , and the best known example is $(3, 4, 5)$. The following list comprehension creates the Pythagorean triples:

```
>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x**2 + y**2 == z**2]
```

```
[ (3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9,
12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]
>>>
```

Another example: Let A and B be two sets, the cross product (or Cartesian product) of A and B, written $A \times B$, is the set of all pairs wherein the first element is a member of the set A and the second element is a member of the set B.

Mathematical definition:

$$A \times B = \{(a, b) : a \text{ belongs to } A, b \text{ belongs to } B\}.$$

It's easy to be accomplished in Python:

```
>>> colours = [ "red", "green", "yellow", "blue" ]
>>> things = [ "house", "car", "tree" ]
>>> coloured_things = [ (x,y) for x in colours for y in things ]
>>> print(coloured_things)
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green',
'house'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house'),
('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue',
'car'), ('blue', 'tree')]
>>>
```

GENERATOR COMPREHENSION

Generator comprehensions were introduced with Python 2.6. They are simply like a list comprehension but with parentheses - round brackets - instead of (square) brackets around it. Otherwise, the syntax and the way of working is like list comprehension, but a generator comprehension returns a generator instead of a list.

```
>>> x = (x **2 for x in range(20))
>>> print(x)
at 0xb7307aa4>
>>> x = list(x)
>>> print(x)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361]
```

A MORE DEMANDING EXAMPLE

Calculation of the prime numbers between 1 and 100 using the sieve of Eratosthenes:

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 100, i)]
>>> primes = [x for x in range(2, 100) if x not in noprimes]
>>> print(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
>>>
```

We want to bring the previous example into more general form, so that we can calculate the list of prime numbers up to an arbitrary number n:

```
>>> from math import sqrt
>>> n = 100
>>> sqrt_n = int(sqrt(n))
>>> no_primes = [j for i in range(2, sqrt_n+1) for j in range(i*2,
n, i)]
```

If we have a look at the content of no_primes, we can see that we have a problem. There are lots of double entries contained in this list:

```
>>> no_primes
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70,
72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 6, 9, 12,
15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63,
66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99, 8, 12, 16, 20, 24,
28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88, 92,
96, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,
90, 95, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96,
14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 16, 24, 32, 40,
48, 56, 64, 72, 80, 88, 96, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99]
>>>
```

The solution to this unbearable problem is easier than you may think. It's just a matter of changing square brackets into braces, or in other words: We will use set comprehension.

SET COMPREHENSION

A set comprehension is similar to a list comprehension, but returns a set and not a list. Syntactically, we use curly brackets instead of square brackets to create a set. Set comprehension is the right functionality to solve our problem from the previous subsection. We are able to create the set of non primes without doublets:

```
>>> from math import sqrt
>>> n = 100
>>> sqrt_n = int(sqrt(n))
>>> no_primes = {j for i in range(2, sqrt_n+1) for j in range(i*2,
n, i)}
>>> no_primes
{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28,
30, 32, 33, 34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51,
52, 54, 55, 56, 57, 58, 60, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74,
75, 76, 77, 78, 80, 81, 82, 84, 85, 86, 87, 88, 90, 91, 92, 93, 94,
```

```

95, 96, 98, 99}
>>> primes = {i for i in range(2, n) if i not in no_primes}
>>> print(primes)
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97}
>>>

```

RECURSIVE FUNCTION TO CALCULATE THE PRIMES

The following Python script uses a recursive function to calculate the prime numbers. It incorporates the fact that it is enough to examine the multiples of the prime numbers up to the square root of n:

```

from math import sqrt
def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return []
    else:
        p = primes(int(sqrt(n)))
        no_p = {j for i in p for j in range(i*2, n+1, i)}
        p = {x for x in range(2, n + 1) if x not in no_p}
    return p

for i in range(1,50):
    print(i, primes(i))

```

DIFFERENCES BETWEEN VERSION 2.X AND 3.X

In Python 2, the loop control variable is not local, i.e. it can change another variable of that name outside of the list comprehension, as we can see in the following example:

```

>>> x = "This value will be changed in the list comprehension"
>>> res = [x for x in range(3)]
>>> res
[0, 1, 2]
>>> x
2
>>> res = [i for i in range(5)]
>>> i
4
>>>

```

Guido van Rossum referred to this effect as "one of Python's 'dirty little secrets' for years".¹ The reason for doing this was efficiency. "It started out as an intentional compromise to make list comprehensions blindingly fast, and while it was not a common pitfall for beginners, it definitely stung people occasionally."²

This "dirty little secret" is fixed in Python3, as you can see in the following code:

```
$ python3
Python 3.2 (r32:88445, Mar 25 2011, 19:28:28)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> x = "Python 3 fixed the dirty little secret"
>>> res = [x for x in range(3)]
>>> print(res)
[0, 1, 2]
>>> x
'Python 3 fixed the dirty little secret'
>>>
```

FOOTNOTES:

¹ Guido van Rossum: [From List Comprehensions to Generator Expressions](#)

² dto.

GENERATORS

INTRODUCTION

An iterator can be seen as a pointer to a container, e.g. a list structure that can iterate over all the elements of this container. The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object. In some object oriented programming languages, like Perl, Java and Python, iterators are implicitly available and can be used in foreach loops, corresponding to for loops in Python.



Generators are a special kind of function, which enable us to implement or generate iterators.

Iterators are a fundamental concept of Python.

Mostly, iterators are implicitly used, like in the for-loop of Python. We demonstrate this in the following example. We are iterating over a list, but you shouldn't be mistaken: A list is not an iterator, but it can be used like an iterator:

```
>>> cities = ["Paris", "Berlin", "Hamburg", "Frankfurt", "London",
    "Vienna", "Amsterdam", "Den Haag"]
>>> for location in cities:
...     print("location: " + location)
...
location: Paris
location: Berlin
location: Hamburg
location: Frankfurt
location: London
location: Vienna
location: Amsterdam
location: Den Haag
>>>
```

What is really happening, when you use an iterable like a string, a list, or a tuple, inside of a for loop is the following: The function "iter" is called on the iterable. The return value of

iter is an iterable. We can iterate over this iterable with the next function until the iterable is exhausted and returns a StopIteration exception:

```
>>> expertises = ["Novice", "Beginner", "Intermediate",
"Proficient", "Experienced", "Advanced"]
>>> expertises_iterator = iter(expertises)
>>> next(expertises_iterator)
'Novice'
>>> next(expertises_iterator)
'Beginner'
>>> next(expertises_iterator)
'Intermediate'
>>> next(expertises_iterator)
'Proficient'
>>> next(expertises_iterator)
'Experienced'
>>> next(expertises_iterator)
'Advanced'
>>> next(expertises_iterator)
Traceback (most recent call last):
  File "", line 1, in
StopIteration
```

Internally, the for loop also calls the next function and terminates, when it gets StopIteration.

We can simulate this iteration behavior of the for loop in a while loop: You might have noticed that there is something missing in our program: We have to catch the "Stop Iteration" exception:

```
other_cities = ["Strasbourg", "Freiburg", "Stuttgart",
                 "Vienna / Wien", "Hannover", "Berlin",
                 "Zurich"]

city_iterator = iter(other_cities)
while city_iterator:
    try:
        city = next(city_iterator)
        print(city)
    except StopIteration:
        break
```

We get the following output from this program:

```
Strasbourg
Freiburg
Stuttgart
Vienna / Wien
Hannover
Berlin
Zurich
```

The sequential base types as well as the majority of the classes of the standard library of Python support iteration. The dictionary data type dict supports iterators as well. In this

case the iteration runs over the keys of the dictionary:

```
>>> capitals = { "France":"Paris", "Netherlands":"Amsterdam",
    "Germany":"Berlin", "Switzerland":"Bern", "Austria":"Vienna"}
>>> for country in capitals:
...     print("The capital city of " + country + " is " +
capitals[country])
...
The capital city of Switzerland is Bern
The capital city of Netherlands is Amsterdam
The capital city of Germany is Berlin
The capital city of France is Paris
>>>
```

Off-topic: Some readers may be confused to learn from our example that the capital of the Netherlands is not Den Haag (The Hague) but Amsterdam. Amsterdam is the capital of the Netherlands according to the constitution, even though the Dutch parliament and the Dutch government are situated in The Hague, as well as the Supreme Court and the Council of State.

GENERATORS

On the surface generators in Python look like functions, but there is both a syntactic and a semantic difference. One distinguishing characteristic is the yield statements. The yield statement turns a functions into a generator. A generator is a function which returns a generator object. This generator object can be seen like a function which produces a sequence of results instead of a single object. This sequence of values is produced by iterating over it, e.g. with a for loop. The values, on which can be iterated, are created by using the yield statement. The value created by the yield statement is the value following the yield keyword. The execution of the code stops when a yield statement has been reached. The value behind the yield will be returned. The execution of the generator is interrupted now. As soon as "next" is called again on the generator object, the generator function will resume execution right after the yield statement in the code, where the last call exited. The execution will continue in the state in which the generator was left after the last yield. This means that all the local variables still exists, because they are automatically saved between calls. This is a fundamental difference to functions: functions always start their execution at the beginning of the function body, regardless where they had left in previous calls. They don't have any static or persistent values. There may be more than one yield statement in the code of a generator or the yield statement might be inside the body of a loop. If there is a return statement in the code of a generator, the execution will stop with a StopIteration exception error if this code is executed by the Python interpreter. The word "generator" is sometimes ambiguously used to mean both the generator function itself and the objects which are generated by a generator.

Everything which can be done with a generator can also be implemented with a class based iterator as well. But the crucial advantage of generators consists in automatically creating the methods `__iter__()` and `next()`.

Generators provide a very neat way of producing data which is huge or infinite.

The following is a simple example of a generator, which is capable of producing various city names:

```
def city_generator():
    yield("London")
    yield("Hamburg")
    yield("Konstanz")
    yield("Amsterdam")
    yield("Berlin")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

It's possible to create a generator object with this generator, which generates all the city names, one after the other:

```
>>> from city_generator import city_generator
>>> city = city_generator()
>>> print(next(city))
London
>>> print(next(city))
Hamburg
>>> print(next(city))
Konstanz
>>> print(next(city))
Amsterdam
>>> print(next(city))
Berlin
>>> print(next(city))
Zurich
>>> print(next(city))
Schaffhausen
>>> print(next(city))
Stuttgart
>>> print(next(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

As we can see, we have generated an iterator `x` in the interactive shell. Every call of the method `next()` returns another city. After the last city, i.e. Stuttgart, has been created, another call of `next(x)` raises an error, saying that the iteration has stopped, i.e. "StopIteration".

Can we send a reset to an iterator is a frequently asked question, so that it can start the iteration all over again. There is no reset, but it's possible to create another generator. This can be done e.g. by having the statement "`x = city_generator()`" again.

Though at first sight the `yield` statement looks like the return statement of a function, we

can see in this example that there is a big difference. If we had a return statement instead of a yield in the previous example, it would be a function. But this function would always return "London" and never any of the other cities, i.e. "Hamburg", "Konstanz", "Amsterdam", "Berlin", "Zurich", "Schaffhausen", and "Stuttgart"

METHOD OF OPERATION

As we have elaborated in the introduction of this chapter, the generators offer a comfortable method to generate iterators, and that's why they are called generators.

Method of working:

- A generator is called like a function. Its return value is an iterator, i.e. a generator object. The code of the generator will not be executed at this stage.
- The iterator can be used by calling the next method. The first time the execution starts like a function, i.e. the first line of code within the body of the iterator. The code is executed until a yield statement is reached.
- yield returns the value of the expression, which is following the keyword yield. This is like a function, but Python keeps track of the position of this yield and the state of the local variables is stored for the next call. At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call.
- The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value.

We will illustrate this behaviour in the following example, in which we define a generator which generates an iterator for all the Fibonacci numbers.

The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of filius Bonacci, "son of Bonaccio"). In his textbook Liber Abaci, which appeared in the year 1202) he had an exercise about the rabbits and their breeding: It starts with a newly-born pair of rabbits, i.e. a male and a female animal. It takes one month until they can mate. At the end of the second month the female gives birth to a new pair of rabbits. Now let's suppose that every female rabbit will bring forth another pair of rabbits every month after the end of the first month. We have to mention that Fibonacci's rabbits never die. The question is how large the population will be after a certain period of time.

This produces a sequence of numbers: 0,1,1,2,3,5,8,13

This sequence can be defined in mathematical terms like this:

$$F_n = F_{n-1} + F_{n-2}$$

with the seed values:

$$F_0 = 0 \text{ and } F_1 = 1$$

```

def fibonacci(n):
    """ A generator for creating the Fibonacci numbers """
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5)
for x in f:
    print(x, " ", end="")
print()

```

The generator above can be used to create the first n Fibonacci numbers separated by blanks, or better (n+1) numbers because the 0th number is also included.

In the next example we present a version which is capable of returning an endless iterator. We have to take care when we use this iterator that a termination criterion is used:

```

def fibonacci():
    """Generates an infinite sequence of Fibonacci numbers on demand"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

f = fibonacci()

counter = 0
for x in f:
    print(x, " ", end="")
    counter += 1
    if (counter > 10):
        break
print()

```

USING A 'RETURN' IN A GENERATOR

Since Python 3.3, generators can also use return statements, but a generator still needs at least one yield statement to be a generator! A return statement inside of a generator is equivalent to `raise StopIteration()`

Let's have a look at a generator in which we raise StopIteration:

```

>>> def gen():
...     yield 1
...     raise StopIteration(42)
...
>>>
>>> g = gen()

```

```
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "", line 1, in
  File "", line 3, in gen
StopIteration: 42
>>>
```

We demonstrate now that return is equivalent, or "nearly", if we disregard one line of the traceback:

```
>>> def gen():
...     yield 1
...     return 42
...
>>>
>>> g = gen()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "", line 1, in
StopIteration: 42
>>>
```

SEND METHOD /COROUTINES

Generators can not only send objects but also receive objects. Sending a message, i.e. an object, into the generator can be achieved by applying the send method to the generator object. Be aware of the fact that send both sends a value to the generator and returns the value yielded by the generator. We will demonstrate this behavior in the following simple example of a coroutine:

```
>>> def simple_coroutine():
...     print("coroutine has been started!")
...     x = yield
...     print("coroutine received: ", x)
...
>>> cr = simple_coroutine()
>>> cr

>>> next(cr)
coroutine has been started!
>>> cr.send("Hi")
coroutine received:  Hi
Traceback (most recent call last):
  File "", line 1, in
StopIteration
>>>
```

We had to call next on the generator first, because the generator needed to be started. Using send to a generator which hasn't been started leads to an exception:

```
>>> cr = simple_coroutine()
>>> cr.send("Hi")
Traceback (most recent call last):
  File "", line 1, in
TypeError: can't send non-None value to a just-started generator
>>>
```

To use the send method the generator has to wait at a yield statement, so that the data sent can be processed or assigned to the variable on the left side. What we haven't said so far: A next call also sends and receives. It always sends a None object. The values sent by "next" and "send" are assigned to a variable within the generator: this variable is called "message" in the following example. We called the generator `infinite_looper`, because it takes a sequential data objects and creates an iterator, which is capable of looping forever over the object, i.e. it starts again with the first element after having delivered the last object. By sending an index to the iterator, we can continue at an arbitrary position.

```
def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        message = yield objects[count]
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We demonstrate how to use this generator in the following interactive session, assuming that the generator is saved in a file called `generator_decorator.py`:

```
>>> x = infinite_looper("A string with some words")
>>> next(x)
'A'
>>> x.send(9)
'w'
>>> x.send(10)
'i'
>>>
```

THE THROW METHOD

The `throw()` method raises an exception at the point where the generator was paused, and returns the next value yielded by the generator. It raises `StopIteration` if the generator exits without yielding another value. The generator has to catch the passed-in exception, otherwise the exception will be propagated to the caller. The `infinite_looper` from our previous example keeps yielding the elements of the sequential data, but we don't have any information about the index or the state of the variable "count". We can get this information

by throwing an exception with the "throw" method. We catch this exception inside of the generator and print the value of "count":

```
def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        try:
            message = yield objects[count]
        except Exception:
            print("index: " + str(count))
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We can use it like this:

```
>>> from generator_throw import infinite_looper
>>> looper = infinite_looper("Python")
>>> next(looper)
'P'
>>> next(looper)
'Y'
>>> looper.throw(Exception)
index: 1
't'
>>> next(looper)
'h'
```

We can improve the previous example by defining our own exception class `StateOfGenerator`:

```
class StateOfGenerator(Exception):
    def __init__(self, message=None):
        self.message = message

def infinite_looper(objects):
    count = 0
    while True:
        if count >= len(objects):
            count = 0
        try:
            message = yield objects[count]
        except StateOfGenerator:
            print("index: " + str(count))
        if message != None:
            count = 0 if message < 0 else message
        else:
            count += 1
```

We can use the previous generator like this:

```
>>> from generator_throw import infinite_looper, StateOfGenerator
>>> looper = infinite_looper("Python")
```

```
>>> next(looper)
'P'
>>> next(looper)
'y'
>>> looper.throw(StateOfGenerator)
index: 1
't'
>>> next(looper)
'h'
>>>
```

DECORATING GENERATORS

There is one problem with our approach, we cannot start the iterator by sending directly an index to it. Before we can do this, we need to use the next function to start the iterator and advance it to the yield statement. We will write a decorator now, which can be used to make a decorator ready, by advancing it automatically at creation time to the yield statement. This way, it will be possible to use the send method directly after initialisation of a generator object.

```
from functools import wraps

def get_ready(gen):
    """
    Decorator: gets a generator gen ready
    by advancing to first yield statement
    """
    @wraps(gen)
    def generator(*args, **kwargs):
        g = gen(*args, **kwargs)
        next(g)
        return g
    return generator

@get_ready
def infinite_looper(objects):
    count = -1
    message = yield None
    while True:
        count += 1
        if message != None:
            count = 0 if message < 0 else message
        if count >= len(objects):
            count = 0
        message = yield objects[count]

x = infinite_looper("abcdef")
print(next(x))
print(x.send(4))
print(next(x))
print(next(x))
print(x.send(5))
print(next(x))
```

This program returns the following results:

```
a
e
f
a
f
a
```

You might have noticed that we have changed the generator `infinite_looper` a little bit as well.

YIELD FROM

"yield from" is available since Python 3.3!

The `yield from <expr>` statement can be used inside the body of a generator. `<expr>` has to be an expression evaluating to an iterable, from which an iterator will be extracted.

The iterator is run to exhaustion, i.e. until it encounters a `StopIteration` exception. This iterator yields and receives values to or from the caller of the generator, i.e. the one which contains the `yield from` statement.

We can learn from the following example by looking at the two generators 'gen1' and 'gen2' that `yield from` is substituting the for loops of 'gen1':

```
def gen1():
    for char in "Python":
        yield char
    for i in range(5):
        yield i

def gen2():
    yield from "Python"
    yield from range(5)

g1 = gen1()
g2 = gen2()
print("g1: ", end=", ")
for x in g1:
    print(x, end=", ")
print("\ng2: ", end=", ")
for x in g2:
    print(x, end=", ")
print()
```

We can see from the output that both generators are the same:

```
g1: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
g2: , P, y, t, h, o, n, 0, 1, 2, 3, 4,
```

The benefit of a `yield from` statement can be seen as a way to split a generator into multiple generators. That's what we have done in our previous example and we will demonstrate this more explicitly in the following example:

```

def cities():
    for city in ["Berlin", "Hamburg", "Munich", "Freiburg"]:
        yield city

def squares():
    for number in range(10):
        yield number ** 2

def generator_all_in_one():
    for city in cities():
        yield city
    for number in squares():
        yield number

def generator_splitted():
    yield from cities()
    yield from squares()

lst1 = [el for el in generator_all_in_one()]
lst2 = [el for el in generator_splitted()]
print(lst1 == lst2)

```

The previous code returns `True` because the generators `generator_all_in_one` and `generator_splitted` yield the same elements. This means that if the `<expr>` from the `yield from` is another generator, the effect is the same as if the body of the sub-generator were inlined at the point of the `yield from` statement. Furthermore, the subgenerator is allowed to execute a return statement with a value, and that value becomes the value of the `yield from` expression. We demonstrate this with the following little script:

```

def subgenerator():
    yield 1
    return 42

def delegating_generator():
    x = yield from subgenerator()
    print(x)

for x in delegating_generator():
    print(x)

```

The above code returns the following code:

```

1
42

```

The full semantics of the `yield from` expression is described in six points in "*PEP 380 -- Syntax for Delegating to a Subgenerator*" in terms of the generator protocol:

- Any values that the iterator yields are passed directly to the caller.
- Any values sent to the delegating generator using `send()` are passed directly to the iterator. If the sent value is `None`, the iterator's `__next__()` method is called. If the sent value is not `None`, the iterator's `send()` method is called. If the call raises

`stopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.

- Exceptions other than `GeneratorExit` thrown into the delegating generator are passed to the `throw()` method of the iterator. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
- If a `GeneratorExit` exception is thrown into the delegating generator, or the `close()` method of the delegating generator is called, then the `close()` method of the iterator is called if it has one. If this call results in an exception, it is propagated to the delegating generator. Otherwise, `GeneratorExit` is raised in the delegating generator.
- The value of the `yield from` expression is the first argument to the `StopIteration` exception raised by the iterator when it terminates.
- `return expr` in a generator causes `StopIteration(expr)` to be raised upon exit from the generator.

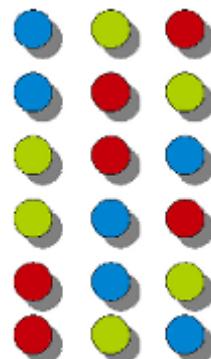
RECURSIVE GENERATORS

Like functions generators can be recursively programmed. The following example is a generator to create all the permutations of a given list of items.

For those who don't know what permutations are, we have a short introduction:

Formal Definition:

A permutation is a rearrangement of the elements of an ordered list.
In other words: Every arrangement of n elements is called a permutation.



In the following lines we show you all the permutations of the letter a, b and c:

```
a b c
a c b
b a c
b c a
c a b
c b a
```

The number of permutations on a set of n elements is given by $n!$

$$n! = n * (n-1) * (n-2) \dots 2 * 1$$

$n!$ is called the factorial of n.

The permutation generator can be called with an arbitrary list of objects. The iterator returned by this generator generates all the possible permutations:

```
def permutations(items):
    n = len(items)
    if n==0: yield []
    else:
        for i in range(len(items)):
            for cc in permutations(items[:i]+items[i+1:]):
                yield [items[i]]+cc

for p in permutations(['r','e','d']): print(''.join(p))
for p in permutations(list("game")): print(''.join(p) + ", ",
end="")
```

The previous example can be hard to understand for newbies. As often, Python offers a convenient solution. We need the module `itertools` for this purpose. `Itertools` is a very handy tool to create and operate on iterators.

Creating permutations with `itertools`:

```
>>> import itertools
>>> perms = itertools.permutations(['r','e','d'])
>>> perms
<itertools.permutations object at 0x7fb0da3e4a70>
>>> list(perms)
[('r', 'e', 'd'), ('r', 'd', 'e'), ('e', 'r', 'd'), ('e', 'd', 'r'),
 ('d', 'r', 'e'), ('d', 'e', 'r')]
>>>
```

The term "permutations" can sometimes be used in a weaker meaning. Permutations can denote in this weaker meaning a sequence of elements, where each element occurs just once, but without the requirement to contain all the elements of a given set. So in this sense (1,3,5,2) is a permutation of the set of digits {1,2,3,4,5,6}. We can build, for example, all the sequences of a fixed length k of elements taken from a given set of size n with $k \leq n$.

These are all the 3-permutations of the set {"a","b","c","d"}:

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

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

These atypical permutations are also known as **sequences without repetition**. By using this term we can avoid confusion with the term "permutation". The number of such k-permutations of n is denoted by $P_{n,k}$ and its value is calculated by the product:

$$n \cdot (n - 1) \cdot \dots \cdot (n - k + 1)$$

By using the factorial notation, the above expression can be written as:

$$P_{n,k} = n! / (n - k)!$$

A generator for the creation of k-permutations of n objects looks very similar to our previous permutations generator:

```
def k_permutations(items, n):
    if n==0:
        yield []
    else:
        for item in items:
            for kp in k_permutations(items, n-1):
                if item not in kp:
                    yield [item] + kp

for kp in k_permutations("abcd", 3):
    print(kp)
```

The above program returns the following k-permutations:

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

```
[ 'c', 'b', 'd']
[ 'c', 'd', 'a']
[ 'c', 'd', 'b']
[ 'd', 'a', 'b']
[ 'd', 'a', 'c']
[ 'd', 'b', 'a']
[ 'd', 'b', 'c']
[ 'd', 'c', 'a']
[ 'd', 'c', 'b']
{ 'c', 'd', 'e'}
{ 'a', 'd', 'e'}
{ 'f', 'd', 'e'}
{ 'b', 'd', 'e'}
{ 'f', 'c', 'd'}
{ 'c', 'a', 'd'}
{ 'b', 'c', 'd'}
{ 'f', 'a', 'd'}
{ 'b', 'a', 'd'}
{ 'b', 'd', 'f'}
{ 'c', 'a', 'e'}
{ 'f', 'a', 'e'}
{ 'b', 'a', 'e'}
{ 'f', 'c', 'a'}
{ 'b', 'c', 'a'}
{ 'b', 'a', 'f'}
{ 'f', 'c', 'e'}
{ 'b', 'c', 'e'}
{ 'b', 'c', 'f'}
{ 'b', 'f', 'e'}
```

A GENERATOR OF GENERATORS

The second generator of our Fibonacci sequence example generates an iterator, which can theoretically produce all the Fibonacci numbers, i.e. an infinite number. But you shouldn't try to produce all these numbers, as we would do in the following example:

```
list(fibonacci())
```

This will show you very fast the limits of your computer.

In most practical applications, we only need the first n elements of an "endless" iterator. We can use another generator, in our example `firstn`, to create the first n elements of a generator g:

```
def firstn(g, n):
    for i in range(n):
        yield next(g)
```

The following script returns the first 10 elements of the Fibonacci sequence:

```
#!/usr/bin/env python3
def fibonacci():
    """Ein Fibonacci-Zahlen-Generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def firstn(g, n):
    for i in range(n):
        yield next(g)

print(list(firstn(fibonacci(), 10)))
```

The output looks like this:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

EXERCISES

1. Write a generator which computes the running average.
2. Write a generator "trange", which generates a sequence of time tuples from start to stop incremented by step. A time tuple is a 3-tuple of integers: (hours, minutes, seconds)

Example:

```
for time in trange((10, 10, 10), (13, 50, 15), (0, 15, 12) ):
    print(time)
```

will return

```
(10, 10, 10)
(10, 25, 22)
(10, 40, 34)
(10, 55, 46)
(11, 10, 58)
(11, 26, 10)
(11, 41, 22)
(11, 56, 34)
(12, 11, 46)
(12, 26, 58)
(12, 42, 10)
(12, 57, 22)
(13, 12, 34)
(13, 27, 46)
(13, 42, 58)
```

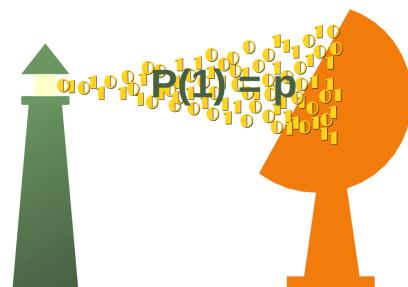
3. Write a version "rtrange" of the previous generator, which can receive message to reset the start value.

4. Write a program, using the newly written generator "trange", to create a file "times_and_temperatures.txt". The lines of this file contain a time in the format hh:mm:ss and random temperatures between 10.0 and 25.0 degrees. The times should be ascending in steps of 90 seconds starting with 6:00:00.

For example:

```
06:00:00 20.1
06:01:30 16.1
06:03:00 16.9
06:04:30 13.4
06:06:00 23.7
06:07:30 23.6
06:09:00 17.5
06:10:30 11.0
```

5. Write a generator with the name "random_ones_and_zeroes", which returns a bitstream, i.e. a zero or a one in every iteration. The probability p for returning a 1 is defined in a variable p . The generator will initialize this value to 0.5. This means that zeroes and ones will be returned with the same probability.



SOLUTIONS TO OUR EXERCISES

```
1. def running_average():
    total = 0.0
    counter = 0
    average = None
    while True:
        term = yield average
        total += term
        counter += 1
```

```

        average = total / counter

ra = running_average()  # initialize the coroutine
next(ra)                 # we have to start the coroutine
for value in [7, 13, 17, 231, 12, 8, 3]:
    out_str = "sent: {val:3d}, new average: {avg:6.2f}"
    print(out_str.format(val=value, avg=ra.send(value)))

2. def trange(start, stop, step):
    """
    trange(stop) -> time as a 3-tuple (hours, minutes, seconds)
    trange(start, stop[, step]) -> time tuple

    start: time tuple (hours, minutes, seconds)
    stop: time tuple
    step: time tuple

    returns a sequence of time tuples from start to stop
    incremented by step
    """
    current = list(start)
    while current < list(stop):
        yield tuple(current)
        seconds = step[2] + current[2]
        min_borrow = 0
        hours_borrow = 0
        if seconds < 60:
            current[2] = seconds
        else:
            current[2] = seconds - 60
            min_borrow = 1
        minutes = step[1] + current[1] + min_borrow
        if minutes < 60:
            current[1] = minutes
        else:
            current[1] = minutes - 60
            hours_borrow = 1
        hours = step[0] + current[0] + hours_borrow
        if hours < 24:
            current[0] = hours
        else:
            current[0] = hours - 24

    if __name__ == "__main__":
        for time in trange((10, 10, 10), (13, 50, 15), (0, 15, 12)):
            print(time)

3. def rtrange(start, stop, step):
    """
    trange(stop) -> time as a 3-tuple (hours, minutes, seconds)

```

```

trange(start, stop[, step]) -> time tuple

start: time tuple (hours, minutes, seconds)
stop: time tuple
step: time tuple

returns a sequence of time tuples from start to stop
incremented by step

The generator can be rest by sending a new "start" value.
"""

current = list(start)
while current < list(stop):
    new_start = yield tuple(current)
    if new_start != None:
        current = list(new_start)
        continue
    seconds = step[2] + current[2]
    min_borrow = 0
    hours_borrow = 0
    if seconds < 60:
        current[2] = seconds
    else:
        current[2] = seconds - 60
        min_borrow = 1
    minutes = step[1] + current[1] + min_borrow
    if minutes < 60:
        current[1] = minutes
    else:
        current[1] = minutes - 60
        hours_borrow = 1
    hours = step[0] + current[0] + hours_borrow
    if hours < 24:
        current[0] = hours
    else:
        current[0] = hours - 24

if __name__ == "__main__":
    ts = rtrange((10, 10, 10), (13, 50, 15), (0, 15, 12) )
    for _ in range(3):
        print(next(ts))

    print(ts.send((8, 5, 50)))
    for _ in range(3):
        print(next(ts))

```

Calling this program will return the following output:

```
(10, 10, 10)
(10, 25, 22)
(10, 40, 34)
(8, 5, 50)
(8, 21, 2)
(8, 36, 14)
(8, 51, 26)
```

```

4. from timerange import trange
   import random

   fh = open("times_and_temperatures.txt", "w")

   for time in trange((6, 0, 0), (23, 0, 0), (0, 1, 30) ):
       random_number = random.randint(100, 250) / 10
       lst = time + (random_number,)
       output = "{:02d}:{:02d}:{:02d} {:4.1f}\n".format(*lst)
       fh.write(output)

```

5. You can find further details and the mathematical background about this exercise in our chapter on **Weighted Probabilities**.

```

import random

def random_ones_and_zeros():
    p = 0.5
    while True:
        x = random.random()
        message = yield 1 if x < p else 0
        if message != None:
            p = message

    x = random_ones_and_zeros()
    next(x) # we are not interested in the return value
    for p in [0.2, 0.8]:
        print("\nWe change the probability to : " + str(p))
        x.send(p)
        for i in range(20):
            print(next(x), end=" ")
    print()

```

We get the following output:

```

We change the probability to : 0.2
0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
We change the probability to : 0.8
1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1

```


ERRORS AND EXCEPTIONS

EXCEPTION HANDLING

An exception is an error that happens during the execution of a program. Exceptions are known to non-programmers as instances that do not conform to a general rule. The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule". Exception handling is a construct in some programming languages to handle or deal with errors automatically. Many programming languages like C++, Objective-C, PHP, Java, Ruby, Python, and many others have built-in support for exception handling.



Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

EXCEPTION HANDLING IN PYTHON

Exceptions handling in Python is very similar to Java. The code, which harbours the risk of an exception, is embedded in a try block. But whereas in Java exceptions are caught by catch clauses, we have statements introduced by an "except" keyword in Python. It's possible to create "custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.

Let's look at a simple example. Assuming we want to ask the user to enter an integer number. If we use a input(), the input will be a string, which we have to cast into an integer. If the input has not been a valid integer, we will generate (raise) a ValueError. We show this in the following interactive session:

```
>>> n = int(input("Please enter a number: "))
Please enter a number: 23.5
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'

```

With the aid of exception handling, we can write robust code for reading an integer from input:

```

while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")

```

It's a loop, which breaks only, if a valid integer has been given.

The example script works like this:

The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left. If an exception occurs, i.e. in the casting of n, the rest of the try block will be skipped and the except clause will be executed. The raised error, in our case a ValueError, has to match one of the names after except. In our example only one, i.e. "ValueError:". After having printed the text of the print statement, the execution does another loop. It starts with a new input().

An example usage could look like this:

```

$ python integer_read.py
Please enter an integer: abc
No valid integer! Please try again ...
Please enter an integer: 42.0
No valid integer! Please try again ...
Please enter an integer: 42
Great, you successfully entered an integer!
$ 

```

MULTIPLE EXCEPT CLAUSES

A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.

Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:

- an IOError
- ValueError

Just in case we have an additional unnamed except clause for an unexpected error:

```
import sys

try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
    # e can be printed directly without using .args:
    # print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

The handling of the IOError in the previous example is of special interest. The except clause for the IOError specifies a variable "e" after the exception name (IOError). The variable "e" is bound to an exception instance with the arguments stored in instance.args. If we call the above script with a non-existing file, we get the message:

```
I/O error(2): No such file or directory
```

And if the file integers.txt is not readable, e.g. if we don't have the permission to read it, we get the following message:

```
I/O error(13): Permission denied
```

An except clause may name more than one exception in a tuple of error names, as we see in the following example:

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred")
except:
    print("An unexpected error occurred")
    raise
```

We want to demonstrate now, what happens, if we call a function within a try block and if an exception occurs inside the function call:

```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
```

```

print("got it :-) ", e)

print("Let's get on")

```

We learn from the above result that the function catches the exception:

```

got it :-) invalid literal for int() with base 10: 'four'
Let's get on

```

We will extend our example now so that the function will catch the exception directly:

```

def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")

```

As we have expected, the exception will be caught inside of the function and not in the callers exception:

```

got it in the function :-) invalid literal for int() with base 10:
'four'
Let's get on

```

We add now a "raise", which generates the ValueError again, so that the exception will be propagated to the caller:

```

def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")

```

We will get the following result:

```

got it in the function :-) invalid literal for int() with base 10:
'four'
got it :-) invalid literal for int() with base 10: 'four'
Let's get on

```

CUSTOM-MADE EXCEPTIONS

It's possible to create Exceptions yourself:

```
>>> raise SyntaxError("Sorry, my fault!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Sorry, my fault!
```

The best or the Pythonic way to do this, consists in defining an exception class which inherits from the Exception class. You will have to go through the chapter on "Object Oriented Programming" to fully understand the following example:

```
class MyException(Exception):
    pass

raise MyException("An exception doesn't always prove the rule!")
```

If you start this program, you will get the following result:

```
$ python3 exception_eigene_klasse.py
Traceback (most recent call last):
  File "exception_eigene_klasse.py", line 4, in <module>
    raise MyException("Was falsch ist, ist falsch!")
__main__.MyException: An exception doesn't always prove the rule!
```

CLEAN-UP ACTIONS (TRY ... FINALLY)

So far the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a finally clause. Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not.

A simple example to demonstrate the finally clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

Let's look at the output of the previous script, if we first input a correct number and after this a string, which is raising an error:

```
bernd@venus:~/tmp$ python finally.py
Your number: 34
There may or may not have been an exception.
The inverse: 0.0294117647059
bernd@venus:~/tmp$ python finally.py
Your number: Python
There may or may not have been an exception.
Traceback (most recent call last):
  File "finally.py", line 3, in <module>
    x = float(input("Your number: "))
ValueError: invalid literal for float(): Python
bernd@venus:~/tmp$
```

COMBINING TRY, EXCEPT AND FINALLY

"finally" and "except" can be used together for the same try block, as can be seen the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

The output of the previous script, if saved as "finally2.py", for various values looks like this:

```
bernd@venus:~/tmp$ python finally2.py
Your number: 37
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: seven
You should have given either an int or a float
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: 0
Infinity
There may or may not have been an exception.
bernd@venus:~/tmp$
```

ELSE CLAUSE

The try ... except statement has an optional else clause. An else block has to be positioned after all the except clauses. An else clause will be executed if the try clause doesn't raise an exception.

The following example opens a file and reads in all the lines into a list called "text":

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
    text = fh.readlines()
    fh.close()
except IOError:
    print('cannot open', file_name)

if text:
    print(text[100])
```

This example receives the file name via a command line argument. So make sure that you call it properly: Let's assume that you saved this program as "exception_test.py". In this case, you have to call it with

```
python exception_test.py integers.txt
```

If you don't want this behaviour, just change the line "file_name = sys.argv[1]" to "file_name = 'integers.txt'".

The previous example is nearly the same as:

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    text = fh.readlines()
    fh.close()

if text:
    print(text[100])
```

The main difference is that in the first case, all statements of the try block can lead to the same error message "cannot open ...", which is wrong, if fh.close() or fh.readlines() raise an error.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Hutchinson adapted for python-course.eu
by Bernd Klein

PYTHON TESTS

ERRORS AND TESTS

Usually, programmers and program developers spend a great deal of their time with debugging and testing. It's hard to give exact percentages, because it highly depends among other factors on the individual programming style, the problems to be solved and of course on the qualification of a programmer. Of course, the programming language is another important factor.



You don't have to program to get pestered by errors, as even the ancient Romans knew. The philosopher Cicero coined more than 2000 years ago an unforgettable aphorism, which is often quoted: "errare humanum est"¹. This aphorism is often used as an excuse for failure. Even though it's hardly possible to completely eliminate all errors in a software product, we should always work ambitiously to this end, i.e. to keep the number of errors minimal.

KINDS OF ERRORS

There are various kinds of errors. During program development there are lots of "small errors", mostly typos. Whether a colon is missing - for example, behind an "if" or an "else" - or the keyword "True" is wrongly written with a lower case "t". These errors are called syntactical errors.²

In most cases, syntactical errors can be easily found, but another type of errors is harder to be solved. A semantic error is syntactically correct code, but the program doesn't behave in the intended way. Imagine somebody wants to increment the value of a variable x by one, but instead of "x += 1" he or she writes "x = 1".

The following longer code example may harbour another semantic error:

```
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
```

```

else:
    print("So what?")

```

We can see two if statements. One nested inside of the other. The code is definitely syntactically correct. But it may be, that the writer of the program only wanted to output "So what?", if the value of the variable x is both greater than 10 and x is not equal to y. In this case, the code should look like this:

```

x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
    else:
        print("So what?")

```

Both code versions are syntactically correct, but one of them violates the intended semantics. Let's look at another example:

```

>>> for i in range(7):
...     print(i)
...
0
1
2
3
4
5
6
>>>

```

The statement ran without raising an exception, so we know that it is syntactically correct. Though it is not possible to decide, if the statement is semantically correct, as we don't know the problem. It may be that the programmer wanted to output the numbers from 1 to 7, i.e. 1,2,...7

In this case, he or she does not properly understand the range function.

So we can divide semantic errors into two categories.

- Errors caused by lack of understanding of a language construct.
- Errors due to logically incorrect code conversion.

UNIT TESTS

This paragraph is about unit tests. As the name implies they are used for testing units or components of the code, typically, classes or functions. The underlying concept is to

simplify the testing of large programming systems by testing "small" units. To accomplish this the parts of a program have to be isolated into independent testable "units". One can define "unit testing" as a method whereby individual units of source code are tested to determine if they meet the requirements, i.e. return the expected output for all possible - or defined - input data. A unit can be seen as the smallest testable part of a program, which are often functions or methods from classes. Testing one unit should be independent from the other units. As a unit is "quite" small, i.e. manageable to ensure complete correctness. Usually, this is not possible for large scale systems like large software programs or operating systems.



MODULE TESTS WITH `__NAME__`

Every module has a name, which is defined in the built-in attribute `__name__`. Let's assume that we have written a module "xyz" which we have saved as "xyz.py". If we import this module with `"import xyz"`, the string "xyz" will be assigned to `__name__`. If we call the file xyz.py as a standalone program, i.e. in the following way,

```
$ python3 xyz.py
```

the value of `__name__` will be the string '`__main__`'.

The following module can be used for calculating fibonacci numbers. But it is not important what the module is doing. We want to demonstrate with it, how it is possible to create a simple module test inside of a module file, - in our case the file "xyz.py", - by using an if statement and checking the value of `__name__`. We check if the module has been started standalone, in which case the value of `__name__` will be '`__main__`'. We assume that you save the following code as "fibonacci.py":

```
""" Fibonacci Module """

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
```

```

        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th
generation """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib

```

It's possible to test this module manually in the interactive Python shell:

```

>>> from fibonacci import fib, fiblist
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fiblist(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fiblist(-8)
[0, 1]
>>> fib(-1)
0
>>> fib(0.5)
Traceback (most recent call last):
  File "", line 1, in
    File "fibonacci.py", line 6, in fib
      for i in range(n):
TypeError: 'float' object cannot be interpreted as an integer
>>>

```

We can see that the functions make only sense, if the input consists of positive integers. The function fib returns 0 for a negative input and fiblist returns always the list [0,1], if the input is a negative integer. Both functions raise a `TypeError` exception, because the `range` function is not defined for floats.

We can test our module by checking the return values for some characteristic calls to `fib()` and `fiblist()`.

So we can add e.g. the following if statement to our module:

```

if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
    print("Test for the fib function was successful!")
else:
    print("The fib function is returning wrong values!")

```

If our program will be called standalone, we see the following output:

```

$ python3 fibonacci.py
Test for the fib function was successful!

```

We will deliberately add an error into our code now.

We change the following line

```
a, b = 0, 1
```

into

```
a, b = 1, 1
```

Principally, the function fib is still calculating the Fibonacci values, but fib(n) is returning the Fibonacci value for the argument "n+1" If we call our changed module, we receive this error message:

```
$ python3 fibonacci.py
"The fib function is returning wrong values!"
```

This approach has a crucial disadvantage. If we import the module, we will get output, saying the test was okay. Something we don't want to see, when we import the module.

```
>>> import fibonacci
Test for the fib function was successful!
```

Apart from being disturbing it is not common practice. Modules should be silent when being imported.

Our module should only print out error messages, warnings or other information, if it is started standalone. If it is imported, it should be silent. The solution for this problem consists in using the built-in attribute `__name__` in a conditional statement. If the module is started standalone, the value of this attribute is "`__main__`". Otherwise the value is the filename of the module without the extension.

Let's rewrite our module in the above mentioned way:

```
""" Fibonacci Module """

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th
generation """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib

if __name__ == "__main__":
    if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
        print("Test for the fib function was successful!")
    else:
        print("The fib function is returning wrong values!")
```

We have squelched our module now. There will be no messages, if the module is imported. This is the simplest and widest used method for unit tests. But it is definitely not the best one.

DOCTEST MODULE

The doctest module is often considered easier to use than the unittest, though the later is more suitable for more complex tests. doctest is a test framework that comes prepackaged with Python. The doctest module searches for pieces of text that look like interactive Python sessions inside of the documentation parts of a module, and then executes (or reexecutes) the commands of those sessions to verify that they work exactly as shown, i.e. that the same results can be achieved. In other words: The help text of the module is parsed, for example, python sessions. These examples are run and the results are compared against the expected value.

Usage of doctest:

To use "doctest" it has to be imported. The part of an interactive Python sessions with the examples and the output has to be copied inside of the docstring the corresponding function.

We demonstrate this way of proceeding with the following simple example. We have slimmed down the previous module, so that only the function fib is left:

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

We now call this module in an interactive Python shell and do some calculations:

```
>>> from fibonacci import fib
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fib(15)
610
>>>
```

We copy the complete session of the interactive shell into the docstring of our function. To start the module doctest we have to call the method testmod(), but only if the module is called standalone. The complete module looks like this now:

```

import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

if __name__ == "__main__":
    doctest.testmod()

```

If we start our module directly like this

```
$ python3 fibonacci_doctest.py
```

we get no output, because everything is okay.

To see how doctest works, if something is wrong, we place an error in our code:
We change again

```
a, b = 0, 1
```

into

```
a, b = 1, 1
```

Now we get the following, if we start our module:

```

$ python3 fibonacci_doctest.py
*****
** File "fibonacci_doctest.py", line 8, in __main__.fib
Failed example:
    fib(0)
Expected:
    0
Got:
    1
*****

```

```
**
File "fibonacci_doctest.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    89
*****
** File "fibonacci_doctest.py", line 14, in __main__.fib
Failed example:
    fib(15)
Expected:
    610
Got:
    987
*****
** 1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```

The output depicts all the calls, which return faulty results. We can see the call with the arguments in the line following "Failed example:". We can see the expected value for the argument in the line following "Expected:". The output shows us the newly calculated value as well. We can find this value behind "Got:"

TEST-DRIVEN DEVELOPMENT (TDD)

In the previous chapters, we tested functions, which we had already been finished. What about testing code you haven't yet written? You think that this is not possible? It is not only possible, it is the underlying idea of test-driven development. In the extreme case, you define tests before you start coding the actual source code. The program developer writes an automated test case which defines the desired "behaviour" of a function. This test case will - that's the idea behind the approach - initially fail, because the code has still to be written.

The major problem or difficulty of this approach is the task of writing suitable tests. Naturally, the perfect test would check all possible inputs and validate the output. Of course, this is generally not always feasible.

We have set the return value of the fib function to 0 in the following example:

```
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively
```

```

>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fib(15)
610
>>>

"""

return 0

if __name__ == "__main__":
    doctest.testmod()

```

It hardly needs mentioning that the function returns except for fib(0) only wrong return values:

```

$ python3 fibonacci_TDD.py
*****
** File "fibonacci_TDD.py", line 10, in __main__.fib
Failed example:
    fib(1)
Expected:
    1
Got:
    0
*****
** File "fibonacci_TDD.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    0
*****
** File "fibonacci_TDD.py", line 14, in __main__.fib
Failed example:
    fib(15)
Expected:
    610
Got:
    0
*****
** 1 items had failures:
    3 of  4 in __main__.fib
***Test Failed*** 3 failures.

```

Now we have to keep on writing and changing the code for the function fib until it passes the test.

This test approach is a method of software development, which is called test-driven development.

UNITTEST

The Python module unittest is a unit testing framework, which is based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework. The module contains the core framework classes that form the basis of the test cases and suites (TestCase, TestSuite and so on), and also a text-based utility class for running the tests and reporting the results (TextTestRunner).

The most obvious difference to the module "doctest" consists in the fact that the test cases of the module "unittest" are not defined inside of the module, which has to be tested. The major advantage is clear: program documentation and test descriptions are separate from each other. The price you have to pay on the other hand consists in an increase of work to create the test cases.

We will use our module fibonacci once more to create a test case with unittest. To this purpose we create a file fibonacci_unittest.py. In this file we have to import unittest and the module which has to be tested, i.e. fibonacci.

Furthermore, we have to create a class with an arbitrary name - we will call it "FibonacciTest" - which inherits from unittest.TestCase. The test cases are defined in this class by using methods. The name of these methods is arbitrary, but has to start with test. In our method "testCalculation" we use the method assertEquals from the class TestCase. assertEquals(first, second, msg = None) checks, if expression "first" is equal to the expression "second". If the two expressions are not equal, msg will be output, if msg is not None.

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def testCalculation(self):
        self.assertEqual(fib(0), 0)
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(5), 5)
        self.assertEqual(fib(10), 55)
        self.assertEqual(fib(20), 6765)

    if __name__ == "__main__":
        unittest.main()
```

If we call this test case, we get the following output:

```
$ python3 fibonacci_unittest.py
.
=====
--
Ran 1 test in 0.000s

OK
```

This is usually the desired result, but we are now interested what happens in the error case. Therefore we will create our previous error again. We change again the well known line:

```
a, b = 0, 1
```

will be changed in

```
a, b = 1, 1
```

Now the test result looks like this:

```
$ python3 fibonacci_unittest.py
F
=====
==
FAIL: testCalculation (__main__.FibonacciTest)
-----
-- 
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 7, in testCalculation
    self.assertEqual(fib(0), 0)
AssertionError: 1 != 0

-----
-- 
Ran 1 test in 0.000s

FAILED (failures=1)
```

The first statement in `testCalculation` has created an exception. The other `assertEqual` calls had not been executed. We correct our error and create a new one. Now all the values will be correct, except if the input argument is 20:

```
def fib(n):
    """ Iterative Fibonacci Function """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    if n == 20:
        a = 42
    return a
```

The output of a test run looks now like this:

```
$ python3 fibonacci_unittest.py
blabal
F
=====
== FAIL: testCalculation (__main__.FibonacciTest)
-----
-- Traceback (most recent call last):
  File "fibonacci_unittest.py", line 12, in testCalculation
    self.assertEqual(fib(20), 6765)
AssertionError: 42 != 6765
-----
-- Ran 1 test in 0.000s
FAILED (failures=1)
```

All the statements of `testCalculation` have been executed, but we haven't seen any output, because everython was okay:

```
self.assertEqual(fib(0), 0)
self.assertEqual(fib(1), 1)
self.assertEqual(fib(5), 5)
```

METHODS OF THE CLASS TESTCASE

We now have a closer look at the class `TestCase`.

Method	Meaning
<code>setUp()</code>	Hook method for setting up the test fixture before exercising it. This method is called before calling the implemented test methods.
<code>tearDown()</code>	Hook method for deconstructing the class fixture after running all tests in the class.
<code>assertEqual(self, first, second, msg=None)</code>	The test fails if the two objects are not equal as determined by the '==' operator.

Method	Meaning
<code>assertAlmostEqual(self, first, second, places=None, msg=None, delta=None)</code>	<p>The test fails if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.</p> <p>Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).</p> <p>If the two objects compare equal then they will automatically compare almost equal.</p>
<code>assertCountEqual(self, first, second, msg=None)</code>	<p>An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.</p> <pre>self.assertEqual(Counter(list(first)), Counter(list(second)))</pre> <p>Example:</p> <p>[0, 1, 1] and [1, 0, 1] compare equal, because the number of ones and zeroes are the same.</p> <p>[0, 0, 1] and [0, 1] compare unequal, because zero appears twice in the first list and only once in the second list.</p>
<code>assertDictEqual(self, d1, d2, msg=None)</code>	Both arguments are taken as dictionaries and they are checked if they are equal.
<code>assertTrue(self, expr, msg=None)</code>	Checks if the expression "expr" is True.
<code>assertGreater(self, a, b, msg=None)</code>	Checks, if $a > b$ is True.
<code>assertGreaterEqual(self, a, b, msg=None)</code>	Checks if $a \geq b$
<code>assertFalse(self, expr, msg=None)</code>	Checks if expression "expr" is False.
<code>assertLess(self, a, b, msg=None)</code>	Checks if $a < b$
<code>assertLessEqual(self, a, b, msg=None)</code>	Checks if $a \leq b$
<code>assertIn(self, member, container, msg=None)</code>	Checks if a in b

Method	Meaning
<code>assertIs(self, expr1, expr2, msg=None)</code>	Checks if "a is b"
<code>assertIsInstance(self, obj, cls, msg=None)</code>	Checks if <code>isinstance(obj, cls)</code> .
<code>assertIsNone(self, obj, msg=None)</code>	Checks if "obj is None"
<code>assert IsNot(self, expr1, expr2, msg=None)</code>	Checks if "a is not b"
<code>assert IsNotNone(self, obj, msg=None)</code>	Checks if obj is not equal to None
<code>assertListEqual(self, list1, list2, msg=None)</code>	Lists are checked for equality.
<code>assertMultiLineEqual(self, first, second, msg=None)</code>	Assert that two multi-line strings are equal.q
<code>assertNotRegexpMatches(self, text, unexpected_reexp, msg=None)</code>	Fails, if the text Text "text" of the regular expression <code>unexpected_reexp</code> matches.
<code>assertTupleEqual(self, tuple1, tuple2, msg=None)</code>	Analogous to <code>assertListEqual</code>

We expand our previous example by a `setUp` and a `tearDown` method:

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def setUp(self):
        self.fib_elems = ( (0,0), (1,1), (2,1), (3,2), (4,3), (5,5) )
        print ("setUp executed!")

    def testCalculation(self):
        for (i,val) in self.fib_elems:
            self.assertEqual(fib(i), val)

    def tearDown(self):
        self.fib_elems = None
        print ("tearDown executed!")

if __name__ == "__main__":
    unittest.main()
```

A call returns the following results:

```
$ python3 fibonacci_unittest2.py
setUp executed!
tearDown executed!
.
-----
-- 
Ran 1 test in 0.000s

OK
```

Most of the TestCase methods have an optional parameter "msg". It's possible to return an additional description of an error with "msg".

EXERCISES

1. Exercise:

Can you find a problem in the following code?

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(40)
    102334155
    >>>

    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    doctest.testmod()
```

Answer:

The doctest is okay. The problem is the implementation of the fibonacci function. This recursive approach is "highly" inefficient. You need a lot of patience to wait for the

termination of the test. The number of hours, days or weeks depend on your computer. ☺

Footnotes:

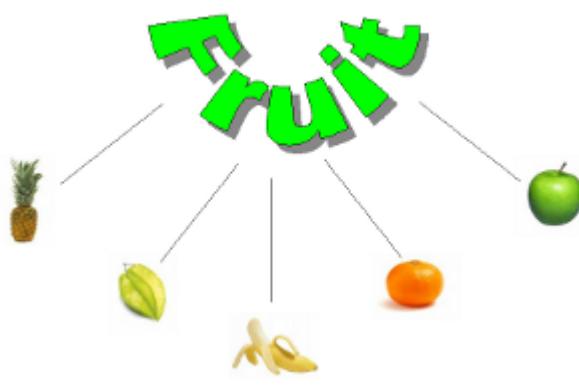
¹The aphorism in full length: "Errare (Errasse) humanum est, sed in errare (errore) perseverare diabolicum." (To err is human, but to persist in it is diabolic")

²In computer science, the syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Writing "if" as "iff" is an example for syntax error, both in programming and in the English language.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

OBJECT-ORIENTED PROGRAMMING

GENERAL INTRODUCTION



Though Python is an object-oriented language without fuss or quibble, we have so far intentionally avoided the treatment of object-oriented programming (OOP) in the previous chapters of our Python tutorial. We skipped OOP, because we are convinced that it is easier and more fun to start learning Python without having to know about all the details of object-oriented programming.

But even though we have avoided OOP, it has nevertheless always been present in the exercises and examples of our course. We used objects and methods from classes without properly explaining their OOP background. In this chapter, we will catch up on what has been missing so far. We will provide an introduction into the principles of object oriented programming in general and into the specifics of the OOP approach of Python. OOP is one of the most powerful tools of Python, but nevertheless you don't have to use it, i.e. you can write powerful and efficient programs without it as well.

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

You will learn to know the four major principles of object-orientation and the way Python deals with them in the next section of this tutorial on object-oriented programming:

- Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

Before we start with the section on the way OOP is used in Python, we want to give you a general idea about object-oriented programming. For this purpose, we would like to draw

your attention to a public library. Let's think about a huge one, like the "British Library" in London or the "New York Public Library" in New York. If it helps, you can imagine the libraries in Paris, Berlin, Ottawa or Toronto¹ as well. Each of these contain an organized collection of books, periodicals, newspapers, audiobooks, films and so on.

Generally, there are two opposed ways of keeping the stock in a library. You can use a "closed access" method that is the stock is not displayed on open shelves. In this system, trained staff brings the books and other publications to the users on demand. Another way of running a library is open-access shelving, also known as "open shelves". "Open" means open to all the users of the library not only specially trained staff. In this case the books are openly displayed. Imperative languages like C could be seen as open-access shelving libraries. The user can do everything. It's up to the user to find the books and to put them back at the right shelf. Even though this is great for the user, it might lead to serious problems in the long run. For example some books will be misplaced, so it's hard to find them again. As you may have guessed already, "closed access" can be compared to object oriented programming. The analogy can be seen like this: The books and other publications, which a library offers, are like the data in an object-oriented program. Access to the books is restricted like access to the data is restricted in OOP. Getting or returning a book is only possible via the staff. The staff functions like the methods in OOP, which control the access to the data. So, the data, - often called attributes, - in such a program can be seen as being hidden and protected by a shell, and it can only be accessed by special functions, usually called methods in the OOP context. Putting the data behind a "shell" is called Encapsulation.
So a library can be regarded as a class and a book is an instance or an object of this class. Generally speaking, an object is defined by a class. A class is a formal description of how an object is designed, i.e. which attributes and methods it has. These objects are called instances as well. The expressions are in most cases used synonymously. A class should not be confused with an object.



OOP IN PYTHON

FIRST-CLASS EVERYTHING

Even though we haven't talked about classes and object orientation in previous chapters, we have worked with classes all the time. In fact, everything is a class in Python. Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." (Blog, The History of Python, February 27, 2009) This means that "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

```
>>> x = 42
>>> type(x)
<class 'int'>
>>> y = 4.34
>>> type(y)
<class 'float'>
>>> def f(x):
...     return x + 1
...
>>> type(f)
<class 'function'>
>>> import math
>>> type(math)
<class 'module'>
>>>
```

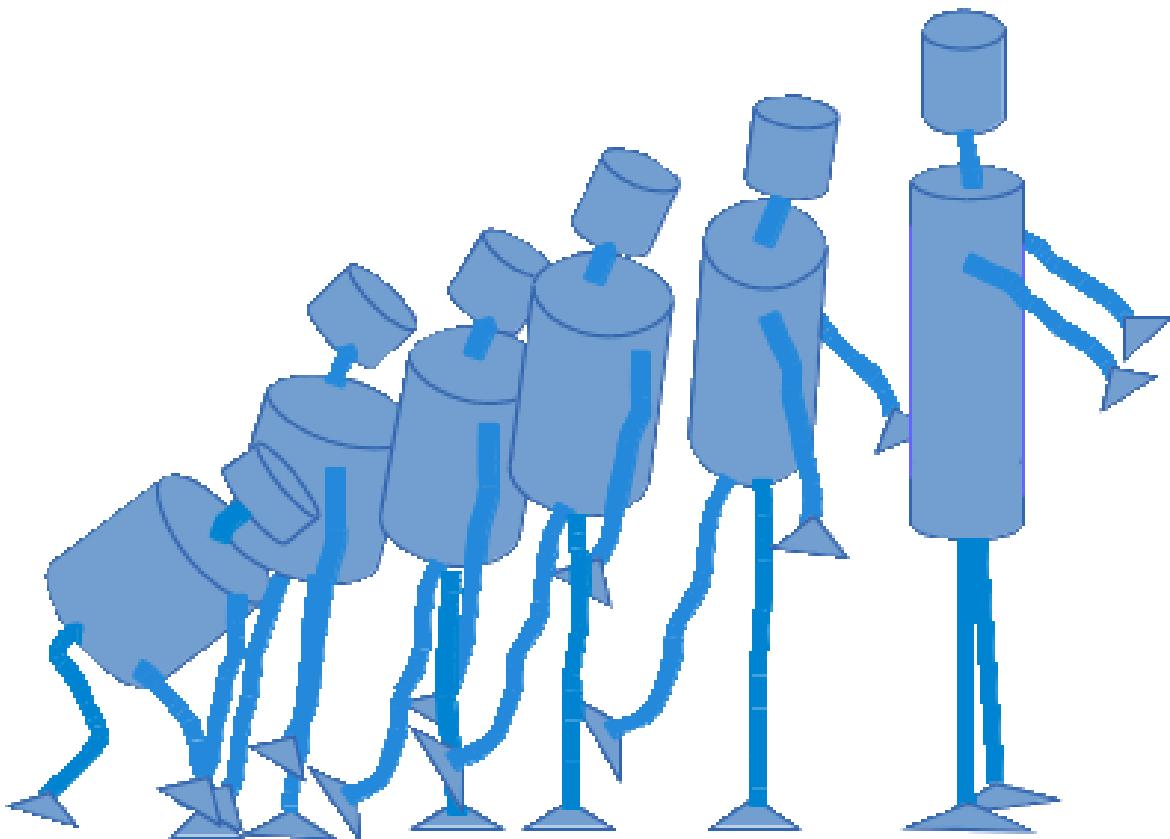
One of the many integrated classes in Python is the list class, which we have quite often used in our exercises and examples. The list class provides a wealth of methods to build lists, to access and change elements, or to remove elements:

```
>>> x = [3, 6, 9]
>>> y = [45, "abc"]
>>> print(x[1])
6
>>> x[1] = 99
>>> x.append(42)
>>> last = y.pop()
>>> print(last)
abc
>>>
```

The variables x and y of the previous example denote two instances of the list class. In simplified terms, we have said so far that "x and y are lists". We will use the terms "object" and "instance" synonymously in the following chapters, as it is often done.²

pop and append of the previous example are methods of the list class. pop returns the most upper (or you might think of it as the "rightest") element of the list and removes this element from the list. We will not explain how Python has implemented lists internally. We don't need this information, because the list class provides us with all the necessary methods to access the data indirectly. This means that the encapsulation details are encapsulated. We will learn about encapsulation later.

A MINIMAL CLASS IN PYTHON



We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

```
class Robot:  
    pass
```

We can realize the fundamental syntactical structure of a class in Python: A class consists of two parts: the header and the body. The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "Robot" in our case. The class name is followed by a listing of other class names, which are classes from which the defined class inherits from. These classes are called superclasses, base classes or sometimes parent classes. If you look at our

example, you will see that this listing of superclasses is not obligatory. You don't have to bother about inheritance and superclasses for the time being. We will introduce them later.

The body of a class consists of an indented block of statements. In our case a single statement, the "pass" statement.

A class object is created, when the definition is left normally, i.e. via the end. This is basically a wrapper around the contents of the namespace created by the class definition.

It's hard to believe, especially for C++ or Java programmers, but we have already defined a complete class with just three words and two lines of code. We are capable of using this class as well:

```
class Robot:  
    pass  
  
if __name__ == "__main__":  
    x = Robot()  
    y = Robot()  
    y2 = y  
    print(y == y2)  
    print(y == x)
```

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y. The output of this example program looks like this:

```
True  
False
```

ATTRIBUTES

Those who have learned already another object-oriented language, will have realized that the terms attributes and properties are usually used synonymously. It may even be used in the definition of an attribute, like Wikipedia does: *"In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such."*

Even in normal English usage the words "attribute" and "property" can be used in some cases as synonyms. Both can have the meaning "An attribute, feature, quality, or characteristic of something or someone". Usually an "attribute" is used to denote a specific

ability or characteristic which something or someone has, like black hair, no hair, or a quick perception, or "her quickness to grasp new tasks". So, think a while about your outstanding attributes. What about your "outstanding properties"? Great, if one of your strong points is your ability to quickly understand and adapt to new situations! Otherwise, you would not learn Python!

Let's get back to Python: We will learn later that properties and attributes are essentially different things in Python. This subsection of our tutorial is about attributes in Python. So far our robots have no attributes. Not even a name, like it is customary for ordinary robots, isn't it? So, let's implement a name attribute. "type designation", "build year" and so on are easily conceivable as further attributes as well.³

Attributes are created inside of a class definition, as we will soon learn. We can dynamically create arbitrary new attributes for existing instances of a class. We do this by joining an arbitrary name to the instance name, separated by a dot ". ". In the following example, we demonstrate this by created an attribute for the name and the build year:

```
>>> class Robot:
...     pass
...
>>> x = Robot()
>>> y = Robot()
>>>
>>> x.name = "Marvin"
>>> x.build_year = "1979"
>>>
>>> y.name = "Caliban"
>>> y.build_year = "1993"
>>>
>>> print(x.name)
Marvin
>>> print(y.build_year)
1993
>>>
```

As we have said before: This is not the way to properly create instance attributes. We introduced this example, because we think that it may help to make the following explanations easier to understand.

If you want to know, what's happening internally: The instances possess dictionaries `__dict__`, which they use to store their attributes and their corresponding values:

```
>>> x.__dict__
{'name': 'Marvin', 'build_year': '1979'}
>>> y.__dict__
{'name': 'Caliban', 'build_year': '1993'}
```

Attributes can be bound to class names as well. In this case, each instance will possess this name as well. Watch out, what happens, if you assign the same name to an instance:

```
>>> class Robot(object):
...     pass
...
>>> x = Robot()
>>> Robot.brand = "Kuka"
>>> x.brand
'Kuka'
>>> x.brand = "Thales"
>>> Robot.brand
'Kuka'
>>> y = Robot()
>>> y.brand
'Kuka'
>>> Robot.brand = "Thales"
>>> y.brand
'Thales'
>>> x.brand
'Thales'
```

If you look at the `__dict__` dictionaries, you can see what's happening.

```
>>> x.__dict__
{'brand': 'Thales'}
>>> y.__dict__
{}
>>>
>>> Robot.__dict__
mappingproxy({'__module__': '__main__', '__weakref__': , '__doc__':
None, '__dict__': , 'brand': 'Thales'})
```

If you try to access `y.brand`, Python checks first, if "brand" is a key of the `y.__dict__` dictionary. If it is not, Python checks, if "brand" is a key of the `Robot.__dict__`. If so, the value can be retrieved.

If an attribute name is not included in either of the dictionary, the attribute name is not defined. If you try to access a non-existing attribute, you will raise an `AttributeError`:

```
>>> x.energy
Traceback (most recent call last):
  File "<stdin>", line 1, in 
AttributeError: 'Robot' object has no attribute 'energy'
>>>
```

By using the function `getattr`, you can prevent this exception, if you provide a default value as the third argument:

```
>>> getattr(x, 'energy', 100)
100
>>>
```

Binding attributes to objects is a general concept in Python. Even function names can be attributed. You can bind an attribute to a function name in the same way, we have done so far to other instances of classes:

```
>>> def f(x):
...     return 42
...
>>> f.x = 42
>>> print(f.x)
42
>>>
```

This can be used as a replacement for the static function variables of C and C++, which are not possible in Python. We use a counter attribute in the following example:

```
def f(x):
    f.counter = getattr(f, "counter", 0) + 1
    return "Monty Python"

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

print(f.counter)
```

If you call this little script, it will output 10.

There may some uncertainty arise at this point. It is possible to assign attributes to most class instances, but this has nothing to do with defining classes. We will see soon how to assign attributes when we define a class.

To properly create instances of classes we also need methods. You will learn in the following subsection of our Python tutorial, how you can define methods.

METHODS

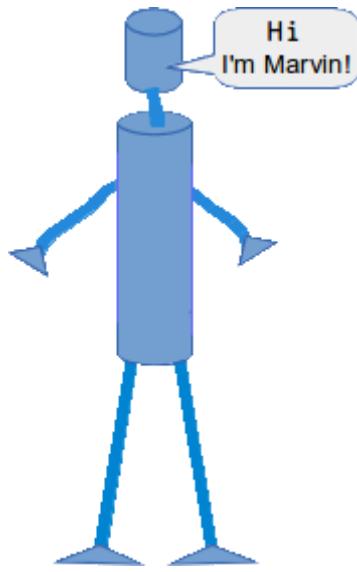
We will demonstrate now, how we can define methods in classes.

Methods in Python are essentially functions in accordance with Guido's saying "first-class everything".

Let's define a function "hi", which takes an object "obj" as an argument and assumes that this object has an attribute "name". We will also define again our basic Robot class:

```
def hi(obj):
    print("Hi, I am " + obj.name + "!")
class Robot:
    pass

x = Robot()
x.name = "Marvin"
hi(x)
```



If we call this code, we get the result

```
Hi, I am Marvin!
```

We will now bind the function „hi“ to a class attribute „say_hi“!

```
def hi(obj):
    print("Hi, I am " + obj.name)

class Robot:
    say_hi = hi

x = Robot()
x.name = "Marvin"
Robot.say_hi(x)
```

"say_hi" is called a method. Usually, it will be called like this:

```
x.say_hi()
```

It is possible to define methods like this, but you shouldn't do it.

The proper way to do it:

- Instead of defining a function outside of a class definition and binding it to a class attribute, we define a method directly inside (indented) of a class definition.
- A method is "just" a function which is defined inside of a class.
- The first parameter is used a reference to the calling instance.
- This parameter is usually called `self`.
- `Self` corresponds to the `Robot` object `x`.

We have seen that a method differs from a function only in two aspects:

- It belongs to a class, and it is defined within a class
- The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "`self`".

As a matter of fact, "`self`" is not a Python keyword. It's just a naming convention! So C++ or Java programmers are free to call it "`this`", but this way they are risking that others might have greater difficulties in understanding their code!

Most other object-oriented programming languages pass the reference to the object (`self`) as a hidden parameter to the methods.

You saw before that the calls `Robot.say_hi(x)`. and "`x.say_hi()`" are equivalent.

"`x.say_hi()`" can be seen as an "abbreviated" form, i.e. Python automatically binds it to the instance name. Besides this "`x.say_hi()`" is the usual way to call methods in Python and in other object oriented languages.

For a Class `C`, an instance `x` of `C` and a method `m` of `C` the following three method calls are equivalent:

- `type(x).m(x, ...)`
- `C.m(x, ...)`
- `x.m(...)`

Before you proceed with the following text, you may mull over the previous example for awhile. Can you figure out, what is wrong in the design?

There is more than one thing about this code, which may disturb you, but the essential problem at the moment is the fact that we create a robot and that after the creation, we shouldn't forget about naming it! If we forget it, `say_hi` will raise an error.

We need a mechanism to initialize an instance right after its creation. This is the `__init__`-method, which we cover in the next section.

THE `__INIT__` METHOD

We want to define the attributes of an instance right after its creation. `__init__` is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to chose another name. `__init__` is one of the so-called magic methods, of which we will get to know some more details later. The `__init__` method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The `__init__` method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

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

We add an `__init__`-method to our robot class:

```
class Robot:

    def __init__(self, name=None):
        self.name = name

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")

x = Robot()
x.say_hi()
y = Robot("Marvin")
y.say_hi()
```

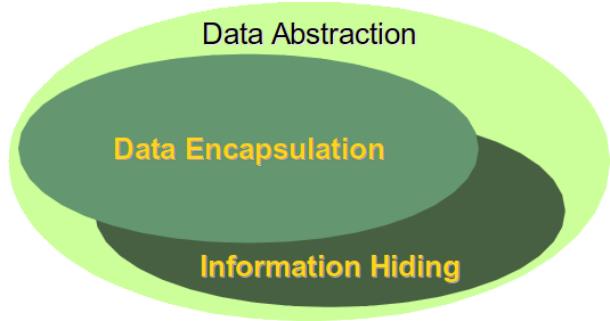
This little program returns the following:

```
Hi, I am a robot without a name
Hi, I am Marvin
```

DATA ABSTRACTION, DATA ENCAPSULATION, AND INFORMATION HIDING

DEFINITIONS OF TERMS

Data Abstraction, Data Encapsulation and Information Hiding are often synonymously used in books and tutorials on OOP. But there is a difference. Encapsulation is seen as the bundling of data with the methods that operate on that data. Information hiding on the other hand is the principle that some internal information or data is "hidden", so that it can't be accidentally changed. Data encapsulation via methods doesn't necessarily mean that the data is hidden. You might be capable of accessing and seeing the data anyway, but using the methods is recommended. Finally, data abstraction is present, if both data hiding and data encapsulation is used. This means data abstraction is the broader term:



Data Abstraction = Data Encapsulation + Data Hiding

Encapsulation is often accomplished by providing two kinds of methods for attributes: The methods for retrieving or accessing the values of attributes are called getter methods. Getter methods do not change the values of attributes, they just return the values. The methods used for changing the values of attributes are called setter methods.

We will define now a Robot class with a Getter and a Setter for the name attribute. We will call them `get_name` and `set_name` accordingly.

```
class Robot:

    def __init__(self, name=None):
        self.name = name

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name
```

```
x = Robot()
x.set_name("Henry")
x.say_hi()
y = Robot()
y.set_name(x.get_name())
print(y.get_name())
```

Hopefully, it will be easy for you to see, that this program prints the following:

```
Hi, I am Henry
Henry
```

Before you go on, you can do a little exercise. You can add an additional attribute "build_year" with Getters and Setters to the Robot class.

```
class Robot:

    def __init__(self,
                 name=None,
                 build_year=None):
        self.name = name
        self.build_year = build_year

    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")
        if self.build_year:
            print("I was built in " + str(self.build_year))
        else:
            print("It's not known, when I was created!")

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_build_year(self, by):
        self.build_year = by

    def get_build_year(self):
        return self.build_year

x = Robot("Henry", 2008)
y = Robot()
y.set_name("Marvin")
x.say_hi()
y.say_hi()
```

The program returns the following output:

```
Hi, I am Henry
I was built in 2008
Hi, I am Marvin
It's not known, when I was created!
```

There is still something wrong with our Robot class. The Zen of Python says: "There should be one-- and preferably only one --obvious way to do it." Our Robot class provides us with two ways to access or to change the "name" or the "build_year" attribute. This can be prevented by using private attributes, which we will explain later.

__STR__- AND __REPR__-METHODS

We will have a short break in our treatise on data abstraction for a quick side-trip. We want to introduce two important magic methods "__str__" and "__repr__", which we will need in future examples. In the course of this tutorial, we have already encountered the __str__ method. We had seen that we can depict various data as string by using the str function, which uses "magically" the internal __str__ method of the corresponding data type. __repr__ is similar. It also produces a string representation.

```
>>> l = ["Python", "Java", "C++", "Perl"]
>>> print(l)
['Python', 'Java', 'C++', 'Perl']
>>> str(l)
"['Python', 'Java', 'C++', 'Perl']"
>>> repr(l)
"['Python', 'Java', 'C++', 'Perl']"
>>> d = {"a":3497, "b":8011, "c":8300}
>>> print(d)
{'a': 3497, 'c': 8300, 'b': 8011}
>>> str(d)
"{'a': 3497, 'c': 8300, 'b': 8011}"
>>> repr(d)
"{'a': 3497, 'c': 8300, 'b': 8011}"
>>> x = 587.78
>>> str(x)
'587.78'
>>> repr(x)
'587.78'
>>>
```

If you apply str or repr to an object, Python is looking for a corresponding method __str__

or `__repr__` in the class definition of the object. If the method does exist, it will be called. In the following example, we define a class A, having neither a `__str__` nor a `__repr__` method. We want to see, what happens, if we use print directly on an instance of this class, or if we apply str or repr to this instance:

```
>>> class A:
...     pass
...
>>> a = A()
>>> print(a)
<__main__.A object at 0xb720a64c>
>>> print(repr(a))
<__main__.A object at 0xb720a64c>
>>> print(str(a))
<__main__.A object at 0xb720a64c>
>>> a
<__main__.A object at 0xb720a64c>
>>>
```

As both methods are not available, Python uses the default output for our object "a".

If a class has a `__str__` method, the method will be used for an instance x of that class, if either the function str is applied to it or if it is used in a print function. `__str__` will not be used, if repr is called, or if we try to output the value directly in an interactive Python shell:

```
>>> class A:
...     def __str__(self):
...         return "42"
...
>>> a = A()

>>> print(repr(a))
<__main__.A object at 0xb720a4cc>
>>> print(str(a))
42
>>> a
<__main__.A object at 0xb720a4cc>
```

Otherwise, if a class has only the `__repr__` method and no `__str__` method, `__repr__` will be applied in the situations, where `__str__` would be applied, if it were available:

```
>>> class A:
...     def __repr__(self):
...         return "42"
...
>>> a = A()
>>> print(repr(a))
42
```

```
>>> print(str(a))
42
>>> a
42
```

A frequently asked question is when to use `__repr__` and when `__str__`. `__str__` is always the right choice, if the output should be for the end user or in other words, if it should be nicely printed. `__repr__` on the other hand is used for the internal representation of an object. The output of `__repr__` should be - if feasible - a string which can be parsed by the python interpreter. The result of this parsing is in an equal object.

This means that the following should be true for an object "o":

`o == eval(repr(o))`

This is shown in the following interactive Python session:

```
>>> l = [3, 8, 9]
>>> s = repr(l)
>>> s
'[3, 8, 9]'
>>> l == eval(s)
True
>>> l == eval(str(l))
True
>>>
```

We show in the following example with the datetime module that eval can only be applied on the strings created by repr:

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str_s = str(today)
>>> eval(str_s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    2014-01-26 17:35:39.215144
    ^
SyntaxError: invalid token
>>> repr_s = repr(today)
>>> t = eval(repr_s)
>>> type(t)
<class 'datetime.datetime'>
>>>
```

We can see that `eval(repr_s)` returns again a `datetime.datetime` object. The String created by `str` can't be turned into a `datetime.datetime` object by parsing it.

We will extend our robot class with a `repr` method. We dropped the other methods to keep

this example simple:

```
class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot('" + self.name + "', " + str(self.build_year)
+ ")"

if __name__ == "__main__":
    x = Robot("Marvin", 1979)

    x_str = str(x)
    print(x_str)
    print("Type of x_str: ", type(x_str))
    new = eval(x_str)
    print(new)
    print("Type of new:", type(new))
```

x_str has the value Robot('Marvin', 1979). eval(x_str) converts it again into a Robot instance.

The script returns the following output:

```
$ python3 robot_class5.py
Robot("Marvin",1979)
Type of x_str: <class 'str'>
Robot("Marvin",1979)
Type of new: <class '__main__.Robot'>
```

Now it's time to extend our class with a user friendly `__str__` method:

```
class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot('" + self.name + "', " + str(self.build_year)
+ ")"

    def __str__(self):
        return "Name: " + self.name + ", Build Year: " +
str(self.build_year)

if __name__ == "__main__":
```

```

x = Robot("Marvin", 1979)

x_str = str(x)
print(x_str)
print("Type of x_str: ", type(x_str))
new = eval(x_str)
print(new)
print("Type of new:", type(new))

```

When we start this program, we can see that it is not possible to convert our string x_str, created via str(x), into a Robot object anymore.

```

$ python3 robot_class6.py
Name: Marvin, Build Year: 1979
Type of x_str: <class 'str'>
Traceback (most recent call last):
  File "robot_class6.py", line 19, in <module>
    new = eval(x_str)
  File "<string>", line 1
    Name: Marvin, Build Year: 1979
      ^
SyntaxError: invalid syntax

```

We show in the following program that x_repr can still be turned into a Robot object:

```

class Robot:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Robot(\"" + self.name + "\", " +
str(self.build_year) + ")"

    def __str__(self):
        return "Name: " + self.name + ", Build Year: " +
str(self.build_year)

if __name__ == "__main__":
    x = Robot("Marvin", 1979)

    x_repr = repr(x)
    print(x_repr, type(x_repr))
    new = eval(x_repr)
    print(new)
    print("Type of new:", type(new))

```

The output looks like this:

```
$ python3 robot_class6b.py
Robot("Marvin",1979) <class 'str'>
Name: Marvin, Build Year: 1979
Type of new: <class '__main__.Robot'>
```

PUBLIC- PROTECTED- AND PRIVATE ATTRIBUTES

Who doesn't know those trigger-happy farmers from films. Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but then he receives a \$50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area.⁴



Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar in the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.

We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under no circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!

Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.



We have the same classification again in object-oriented programming:

- Private attributes should only be used by the owner, i.e. inside of the class definition itself.
- Protected (restricted) Attributes may be used, but at your own risk. Essentially, this means that they should only be used under certain conditions.
- Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course. There are two ways to restrict the access to class attributes:

- First, we can prefix an attribute name with a leading underscore "`_`". This marks the attribute as protected. It tells users of the class not to use this attribute unless, somebody writes a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
- Second, we can prefix an attribute name with two leading underscores "`__`". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside of the class definition itself.⁵

To summarize the attribute types:

Naming Type	Meaning
name	Public
<code>_name</code>	Protected
<code>__name</code>	Private

These attributes can be freely used inside or outside of a class definition.

Protected attributes should not be used outside of the class definition, unless inside of a subclass definition.

This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside of the class definition itself.

We want to demonstrate the behaviour of these attribute types with an example class:

```
class A():

    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
```

We store this class (`attribute_tests.py`) and test its behaviour in the following interactive Python shell:

```
>>> from attribute_tests import A
>>> x = A()
```

```
>>> x.pub
'I am public'
>>> x.pub = x.pub + " and my value can be changed"
>>> x.pub
'I am public and my value can be changed'
>>> x._prot
'I am protected'
>>> x.__priv
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__priv'
>>>
```

The error message is very interesting. One might have expected a message like "`__priv` is private". We get the message "AttributeError: 'A' object has no attribute '`__priv`'" instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task consists in rewriting our Robot class. Though we have Getter and Setter methods for the name and the build_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of `self.name` and `self.build_year` by `self.__name` and `self.__build_year`.

The listing of our revised class:

```
class Robot:

    def __init__(self, name=None, build_year=2000):
        self.__name = name
        self.__build_year = build_year

    def say_hi(self):
        if self.__name:
            print("Hi, I am " + self.__name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_build_year(self, by):
        self.__build_year = by

    def get_build_year(self):
        return self.__build_year
```

```

def __repr__(self):
    return "Robot('" + self.__name + "', " +
str(self.__build_year) + ")"

def __str__(self):
    return "Name: " + self.__name + ", Build Year: " +
str(self.__build_year)

if __name__ == "__main__":
    x = Robot("Marvin", 1979)
    y = Robot("Caliban", 1943)
    for rob in [x, y]:
        rob.say_hi()
        if rob.get_name() == "Caliban":
            rob.set_build_year(1993)
    print("I was built in the year " + str(rob.get_build_year()))
+ "!")

```

We get the following out, if we call this program:

```

Hi, I am Marvin
I was built in the year 1979!
Hi, I am Caliban
I was built in the year 1993!

```

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

This may look like the following class:

```

class A():

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def GetX(self):
        return self.__x

    def GetY(self):
        return self.__y

    def SetX(self, x):
        self.__x = x

    def SetY(self, y):
        self.__y = y

```

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic Code this way, as you will learn soon.

DESTRUCTOR

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method `__del__`. It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with `__init__` and `__del__`:

```
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print("Robot has been destroyed")

if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

The output of the previous program:

```
$ python3 del_example.py
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
Robot has been destroyed
Robot has been destroyed
```

The usage of the `__del__` method is very problematic. If we change the previous code to

personalize the deletion of a robot, we create an error:

```
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print(self.name + " says bye-bye!")

if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

We get the following output with error messages:

```
$ python3 del_example.py
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
Exception AttributeError: "'Robot' object has no attribute 'name'"
in <bound method Robot.__del__ of <__main__.Robot object at
0xb71da3cc>> ignored
Exception AttributeError: "'Robot' object has no attribute 'name'"
in <bound method Robot.__del__ of <__main__.Robot object at
0xb71da36c>> ignored
```

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

FOOTNOTES:

1

The picture on the right side is taken in the Library of the Court of Appeal for Ontario, located downtown Toronto in historic Osgoode Hall²

"Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value." (excerpt from the official Python Language Reference)

3

"attribute" stems from the Latin verb "attribuere" which means "to associate with"

4

<http://www.wickedlocal.com/x937072506/tJogger-ticketed-for-trespassing-in-the-Winchester-Fells-kicks-back>

5 There is a way to access a private attribute directly. In our example, we can do it like this:

x._Robot__build_year

You shouldn't do this under any circumstances!

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

CLASS AND INSTANCE ATTRIBUTES

CLASS ATTRIBUTES

Instance attributes are owned by the specific instances of a class. This means for two different instances the instance attributes are usually different. You should by now be familiar with this concept which we introduced the previous chapter.

We can also define attributes at the class level. Class attributes are attributes which are owned by the class itself. They will be shared by all the instances of the class. Therefore they have the same value for every instance. We define class attributes outside of all the methods, usually they are placed at the top, right below the class header.



We can see in the following interactive Python session that the class attribute "a" is the same for all instances, in our example "x" and "y". Besides this, we see that we can access a class attribute via an instance or via the class name:

```
>>> class A:
...     a = "I am a class attribute!"
...
>>> x = A()
>>> y = A()
>>> x.a
'I am a class attribute!'
>>> y.a
'I am a class attribute!'
>>> A.a
'I am a class attribute!'
>>>
```

But be careful, if you want to change a class attribute, you have to do it with the notation `ClassName.AttributeName`. Otherwise, you will create a new instance variable. We demonstrate this in the following example:

```
>>> class A:
...     a = "I am a class attribute!"
...
>>> x = A()
>>> y = A()
```

```
>>> x.a = "This creates a new instance attribute for x!"
>>> y.a
'I am a class attribute!'
>>> A.a
'I am a class attribute!'
>>> A.a = "This is changing the class attribute 'a'!"
>>> A.a
"This is changing the class attribute 'a'!"
>>> y.a
"This is changing the class attribute 'a'!"
>>> # but x.a is still the previously created instance variable:
...
>>> x.a
'This creates a new instance attribute for x!'
>>>
```

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

```
>>> x.__dict__
{'a': 'This creates a new instance attribute for x!'}
>>> y.__dict__
{}
>>> A.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!",
 '__dict__': <attribute '__dict__' of 'A' objects>, '__module__':
 '__main__', '__weakref__': <attribute '__weakref__' of 'A' objects>,
 '__doc__': None})
>>> x.__class__.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!",
 '__dict__': <attribute '__dict__' of 'A' objects>, '__module__':
 '__main__', '__weakref__': <attribute '__weakref__' of 'A' objects>,
 '__doc__': None})
>>>
```

EXAMPLE WITH CLASS ATTRIBUTES

Isaac Asimov devised and introduced the so-called "Three Laws of Robotics" in 1942. The appeared in his story "Runaround". His three laws have been picked up by many science fiction writers. As we have started manufacturing robots in Python, it's high time to make sure that they obey Asimov's three laws. As they are the same for every instance, i.e. robot, we will create a class attribute Three_Laws. This attribute is a tuple with the three laws.

```
class Robot:

    Three_Laws = (
        """A robot may not injure a human being or, through inaction, allow
        a human being to come to harm.""",
        """A robot must obey the orders given to it by human beings, except
        where such orders would conflict with the First Law.,""",
```

```
"""A robot must protect its own existence as long as such protection
does not conflict with the First or Second Law."""
)

def __init__(self, name, build_year):
    self.name = name
    self.build_year = build_year

# other methods as usual
```

As we mentioned before, we can access a class attribute via instance or via the class name. You can see in the following that we need no instance:

```
>>> from robot_asimov import Robot
>>> for number, text in enumerate(Robot.Three_Laws):
...     print(str(number+1) + ":\n" + text)
...
1:
A robot may not injure a human being or, through inaction, allow a
human being to come to harm.
2:
A robot must obey the orders given to it by human beings, except
where such orders would conflict with the First Law.,
3:
A robot must protect its own existence as long as such protection
does not conflict with the First or Second Law.
>>>
```

We demonstrate in the following example, how you can count instance with class attributes. All we have to do is

- to create a class attribute, which we call "counter" in our example
- to increment this attribute by 1 every time a new instance will be created
- to decrement the attribute by 1 every time an instance will be destroyed

```
class C:

    counter = 0

    def __init__(self):
        type(self).counter += 1

    def __del__(self):
        type(self).counter -= 1

if __name__ == "__main__":
    x = C()
    print("Number of instances: : " + str(C.counter))
```

```

y = C()
print("Number of instances: : " + str(C.counter))
del x
print("Number of instances: : " + str(C.counter))
del y
print("Number of instances: : " + str(C.counter))

```

Principally, we could have written C.counter instead of type(self).counter, because type(self) will be evaluated to "C" anyway. But we will see later, that type(self) makes sense, if we use such a class as a superclass.

If we start the previous program, we will get the following results:

```

$ python3 counting_instances.py
Number of instances: : 1
Number of instances: : 2
Number of instances: : 1
Number of instances: : 0

```

STATIC METHODS

We used class attributes as public attributes in the previous section. Of course, we can make public attributes private as well. We can do this by adding the double underscore again. If we do so, we need a possibility to access and change these private class attributes. We could use instance methods for this purpose:

```

class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def RobotInstances(self):
        return Robot.__counter

if __name__ == "__main__":
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())

```

This is not a good idea for two reasons: First of all, because the number of robots has

nothing to do with a single robot instance and secondly because we can't inquire the number of robots before we haven't created an instance.

If we try to invoke the method with the class name Robot.RobotInstances(), we get an error message, because it needs an instance as an argument:

```
>>> Robot.RobotInstances()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: RobotInstances() takes exactly 1 argument (0 given)
```

The next idea, which still doesn't solve our problem, consists in omitting the parameter "self":

```
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def RobotInstances():
        return Robot.__counter
```

Now it's possible to access the method via the class name, but we can't call it via an instance:

```
>>> from static_methods2 import Robot
>>> Robot.RobotInstances()
0
>>> x = Robot()
>>> x.RobotInstances()
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
TypeError: RobotInstances() takes no arguments (1 given)
>>>
```

The call "x.RobotInstances()" is treated as an instance method call and an instance method needs a reference to the instance as the first parameter.

So, what do we want? We want a method, which we can call via the class name or via the instance name without the necessity of passing a reference to an instance to it.

The solution consists in static methods, which don't need a reference to an instance. It's easy to turn a method into a static method. All we have to do is to add a line with "@staticmethod" directly in front of the method header. It's the decorator syntax.

You can see in the following example that we can now use our method RobotInstances the way we wanted:

```
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def RobotInstances():
        return Robot.__counter

if __name__ == "__main__":
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())
```

We will get the following output:

```
0
1
2
2
```

CLASS METHODS

Static methods shouldn't be confused with class methods. Like static methods class methods are not bound to instances, but unlike static methods class methods are bound to a class. The first parameter of a class method is a reference to a class, i.e. a class object. They can be called via an instance or the class name.

```
class Robot:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @classmethod
    def RobotInstances(cls):
        return cls, Robot.__counter
```

```

if __name__ == "__main__":
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())

```

The output looks like this:

```

$ python3 static_methods4.py
<class '__main__.Robot', 0>
<class '__main__.Robot', 1>
<class '__main__.Robot', 2>
<class '__main__.Robot', 2>

```

The use cases of class methods:

- they are used in the definition of the so-called factory methods, which we will not cover here.
- They are often used, where we have static methods, which have to call other static methods. To do this, we would have to hard code the class name, if we had to use static methods. This is a problem, if we are in a use case, where we have inherited classes.

The following program contains a fraction class, which is still not complete. If you work with fractions, you need to be capable of reducing fractions, e.g. the fraction 8/24 can be reduced to 1/3. We can reduce a fraction to lowest terms by dividing both the numerator and denominator by the Greatest Common Divisor (GCD).

We have defined a static gcd function to calculate the greatest common divisor of two numbers. the greatest common divisor (gcd) of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the 'GCD of 8 and 24 is 8. The static method "gcd" is called by our class method "reduce" with "cls.gcd(n1, n2)". "CLS" is a reference to "fraction".

```

class fraction(object):

    def __init__(self, n, d):
        self.numerator, self.denominator = fraction.reduce(n, d)

    @staticmethod
    def gcd(a,b):
        while b != 0:
            a, b = b, a%b

```

```

        return a

@classmethod
def reduce(cls, n1, n2):
    g = cls.gcd(n1, n2)
    return (n1 // g, n2 // g)

def __str__(self):
    return str(self.numerator) + '/' + str(self.denominator)

```

Using this class:

```

>>> from fraction1 import fraction
>>> x = fraction(8,24)
>>> print(x)
1/3
>>>

```

We will demonstrate in our last example the usefulness of class methods in inheritance. We define a class "Pets" with a method "about". This class will be inherited in a subclass "Dogs" and "Cats". The method "about" will be inherited as well. We will define the method "about" as a "staticmethod" in our first implementation to show the disadvantage of this approach:

```

class Pets:
    name = "pet animals"

    @staticmethod
    def about():
        print("This class is about {}".format(Pets.name))

class Dogs(Pets):
    name = "'man's best friends' (Frederick II)"

class Cats(Pets):
    name = "cats"

p = Pets()
p.about()
d = Dogs()
d.about()
c = Cats()
c.about()

```

We get the following output:

```
This class is about pet animals!
This class is about pet animals!
This class is about pet animals!
```

Especially, in the case of `c.about()` and `d.about()`, we would have preferred a more specific phrase! The "problem" is that the method "about" doesn't know that it has been called by an instance of the Dogs or Cats class.

We decorate it now with a `classmethod` decorator instead of a `staticmethod` decorator:

```
class Pets:
    name = "pet animals"

    @classmethod
    def about(cls):
        print("This class is about {}".format(cls.name))

class Dogs(Pets):
    name = "'man's best friends' (Frederick II)"

class Cats(Pets):
    name = "cats"

p = Pets()
p.about()

d = Dogs()
d.about()

c = Cats()
c.about()
```

The output is now like we wanted it to be:

```
This class is about pet animals!
This class is about 'man's best friends' (Frederick II) !
This class is about cats!
```


PROPERTIES VS. GETTERS AND SETTERS

PROPERTIES

Getters and setters are used in many object oriented programming languages to ensure the principle of data encapsulation. They are known as mutator methods as well. Data encapsulation - as we have learnt in our [introduction on Object Oriented Programming](#) of our tutorial - is seen as the bundling of data with the methods that operate on these data. These methods are of course the getter for retrieving the data and the setter for changing the data. According to this principle, the attributes of a class are made private to hide and protect them from other code.



Unfortunately, it is widespread belief that a proper Python class should encapsulate private attributes by using getters and setters. As soon as one of these programmers introduces a new attribute, he or she will make it a private variable and creates "automatically" a getter and a setter for this attribute. Such programmers may even use an editor or an IDE, which automatically create getters and setters for all private attributes. These tools even warn the programmer if she or he uses a public attribute! Java programmers will wrinkle their brows, screw up their noses, or even scream with horror when they read the following: The Pythonic way to introduce attributes is to make them public.

We will explain this later. First, we demonstrate in the following example, how we can design a class in a Javaesque way with getters and setters to encapsulate the private attribute "self.__x":

```
class P:  
  
    def __init__(self,x):  
        self.__x = x  
  
    def get_x(self):  
        return self.__x
```

```
def set_x(self, x):
    self.__x = x
```

We can see in the following demo session how to work with this class and the methods:

```
>>> from mutators import P
>>> p1 = P(42)
>>> p2 = P(4711)
>>> p1.get_x()
42
>>> p1.set_x(47)
>>> p1.set_x(p1.get_x() + p2.get_x())
>>> p1.get_x()
4758
>>>
```

What do you think about the expression "p1.set_x(p1.get_x() + p2.get_x())"? It's ugly, isn't it? It's a lot easier to write an expression like the following, if we had a public attribute x:

```
p1.x = p1.x + p2.x
```

Such an assignment is easier to write and above all easier to read than the Javaesque expression.

Let's rewrite the class P in a Pythonic way. No getter, no setter and instead of the private attribute "self.__x" we use a public one:

```
class P:
    def __init__(self, x):
        self.x = x
```

Beautiful, isn't it? Just three lines of code, if we don't count the blank line!

```
>>> from p import P
>>> p1 = P(42)
>>> p2 = P(4711)
>>> p1.x
42
>>> p1.x = 47
>>> p1.x = p1.x + p2.x
>>> p1.x
4758
>>>
```

"But, but, but, but, but ... ", we can hear them howling and screaming, "**But there is NO data ENCAPSULATION!**"

Yes, in this case there is no data encapsulation. We don't need it in this case. The only thing `get_x` and `set_x` in our starting example did was "getting the data through" without doing anything, no checks nothing.

But what happens if we want to change the implementation in the future. This is a serious argument. Let's assume we want to change the implementation like this: The attribute `x` can have values between 0 and 1000. If a value larger than 1000 is assigned, `x` should be set to 1000. Correspondingly, `x` should be set to 0, if the value is less than 0.

It is easy to change our first `P` class to cover this problem. We change the `set_x` method accordingly:

```
class P:

    def __init__(self, x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

The following Python session shows that it works the way we want it to work:

```
>>> from mutators import P
>>> p1 = P(1001)
>>> p1.get_x()
1000
>>> p2 = P(15)
>>> p2.get_x()
15
>>> p3 = P(-1)
>>> p3.get_x()
0
```

But there is a catch: Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
from p import P
p1 = P(42)
p1.x = 1001
```

Our new class means breaking the interface. The attribute `x` is not available anymore. That's why in Java e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.

But Python offers a solution to this problem. The solution is called properties!

The class with a property looks like this:

```
class P:

    def __init__(self, x):
        self.x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

A method which is used for getting a value is decorated with "@property", i.e. we put this line directly in front of the header. The method which has to function as the setter is decorated with "@x.setter". If the function had been called "f", we would have to decorate it with "@f.setter".

Two things are noteworthy: We just put the code line "self.x = x" in the `__init__` method and the property method `x` is used to check the limits of the values. The second interesting thing is that we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". We have learned in a previous chapter of our course that this is not possible. It works here due to the decorating:

```
>>> from p import P
>>> p1 = P(1001)
>>> p1.x
1000
>>> p1.x = -12
>>> p1.x
0
>>>
```

Alternatively, we could have used a different syntax without decorators to define the property. As you can see, the code is definitely less elegant and we have to make sure that we use the getter function in the `__init__` method again:

```
class P:

    def __init__(self, x):
        self.set_x(x)

    def get_x(self):
        return self.__x
```

```

def set_x(self, x):
    if x < 0:
        self.__x = 0
    elif x > 1000:
        self.__x = 1000
    else:
        self.__x = x

x = property(get_x, set_x)

```

There is still another problem in the most recent version. We have now two ways to access or change the value of x: Either by using "p1.x = 42" or by "p1.set_x(42)". This way we are violating one of the fundamentals of Python: "There should be one-- and preferably only one --obvious way to do it." (see [Zen of Python](#))

We can easily fix this problem by turning the getter and the setter method into private methods, which can't be accessed anymore by the users of our class P:

```

class P:

    def __init__(self, x):
        self.__set_x(x)

    def __get_x(self):
        return self.__x

    def __set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(__get_x, __set_x)

```

Even though we fixed this problem by using a private getter and setter, the version with the decorator "@property" is the Pythonic way to do it!

From what we have written so far, and what can be seen in other books and tutorials as well, we could easily get the impression that there is a one-to-one connection between properties (or mutator methods) and the attributes, i.e. that each attribute has or should have its own property (or getter-setter-pair) and the other way around. Even in other object oriented languages than Python, it's usually not a good idea to implement a class like that. The main reason is that many attributes are only internally needed and creating interfaces for the user of the class increases unnecessarily the usability of the class. The possible user

of a class shouldn't be "drowned" with umpteen - of mainly unnecessary - methods or properties!

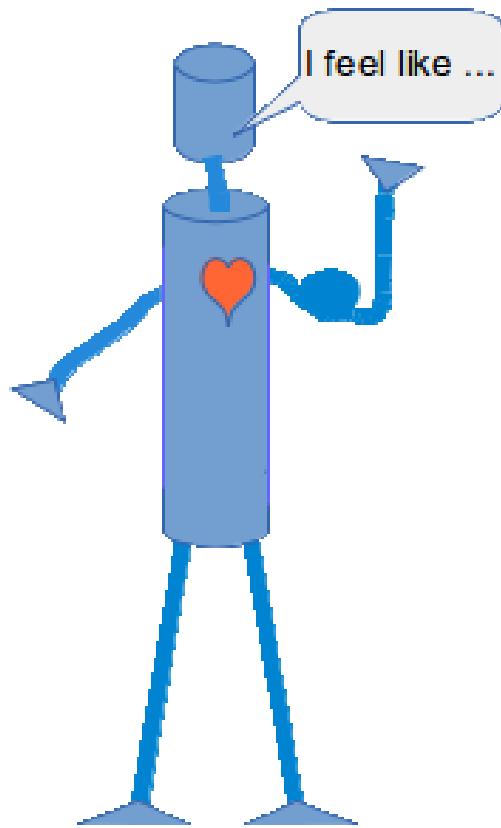
The following example shows a class, which has internal attributes, which can't be accessed from outside. These are the private attributes self.__potential_physical and self.__potential_psychic. Furthermore we show that a property can be deduced from the values of more than one attribute. The property "condition" of our example returns the condition of the robot in a descriptive string. The condition depends on the sum of the values of the psychic and the physical conditions of the robot.

```
class Robot:

    def __init__(self, name,
build_year, lk = 0.5, lp = 0.5 ):
        self.name = name
        self.build_year = build_year
        self.__potential_physical = lk
        self.__potential_psychic = lp

    @property
    def condition(self):
        s = self.__potential_physical + self.__potential_psychic
        if s <= -1:
            return "I feel miserable!"
        elif s <= 0:
            return "I feel bad!"
        elif s <= 0.5:
            return "Could be worse!"
        elif s <= 1:
            return "Seems to be okay!"
        else:
            return "Great!"

if __name__ == "__main__":
    x = Robot("Marvin", 1979, 0.2, 0.4 )
    y = Robot("Caliban", 1993, -0.4, 0.3)
    print(x.condition)
    print(y.condition)
```



PUBLIC INSTEAD OF PRIVATE ATTRIBUTES

Let's summarize the usage of private and public attributes, getters and setters and properties: Let's assume that we are designing a new class and we are pondering about an instance or class attribute "OurAtt", which we need for the design of our class. We have to observe the following issues:

- Will the value of "OurAtt" be needed by the possible users of our class?
- If not, we can or should make it a private attribute.
- If it has to be accessed, we make it accessible as a public attribute
- We will define it as a private attribute with the corresponding property, if and only if we have to do some checks or transformation of the data. (As an example, you can have a look again at our class P, where the attribute has to be in the interval between 0 and 1000, which is ensured by the property "x")
- Alternatively, you could use a getter and a setter, but using a property is the Pythonic way to deal with it!

Let's assume we have defined "OurAtt" as a public attribute. Our class has been successfully used by other users for quite a while. Now comes the point which frightens some traditional OOPistas out of their wits: Imagine "OurAtt" has been used as an integer. Now, our class has to ensure that "OurAtt" has to be a value between 0 and 1000? Without property, this is really a horrible scenario! Due to properties it's easy: We create a property version of "OurAtt".

```
class OurClass:

    def __init__(self, a):
        self.OurAtt = a

    @property
    def OurAtt(self):
        return self.__OurAtt

    @OurAtt.setter
    def OurAtt(self, val):
        if val < 0:
            self.__OurAtt = 0
        elif val > 1000:
            self.__OurAtt =
        else:
            self.__OurAtt =
            val

x = OurClass(10)
print(x.OurAtt)
```



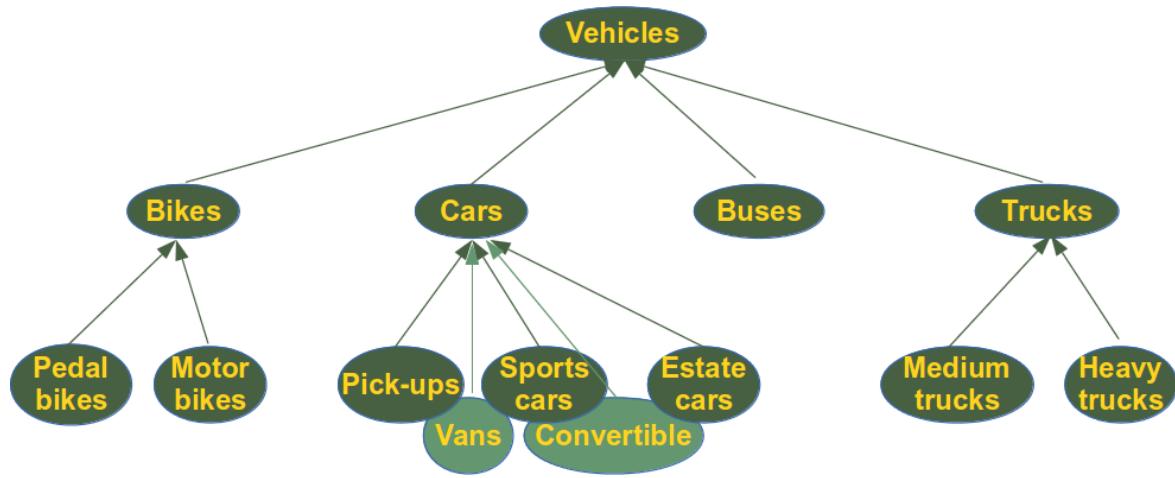
This is great, isn't it: You can start with the simplest implementation imaginable, and you are free to later migrate to a property version without having to change the interface! So properties are not just a replacement for getters and setters!

Something else you might have already noticed: For the users of a class, properties are syntactically identical to ordinary attributes.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

INHERITANCE

INTRODUCTION



Every object-oriented programming language would not be worthy to look at or use, if it weren't to support inheritance. Of course, Python supports inheritance, it even supports multiple inheritance.

Classes can inherit from other classes. A class can inherit attributes and behaviour methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchy relationship between classes. It's similar to relationships or categorizations that we know from real life. Think about vehicles, for example. Bikes, cars, buses and trucks are vehicles. pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

SYNTAX AND SIMPLE INHERITANCE EXAMPLE

We demonstrate inheritance in a very simple example. We create a Person class with the two attributes "firstname" and "lastname". This class has only one method, the Name method, essentially a getter, but we don't have an attribute name. This method is a further example for a "getter", which creates an output by creating it from more than one private attribute. Name returns the concatenation of the first name and the last name of a person, separated by a space. It goes without saying that a useful person class would have additional attributes and further methods.

This chapter of our tutorial is about inheritance, so we need a class, which inherits from Person. So far employees are Persons in companies, even though they may not be treated as such in some firms. If we created an Employee class without inheriting from Person, we would have to define all the attributes and methods in the Employee class again. This means we would create a design and maybe even a data redundancy. With this in mind, we have to let Employee inherit from Person.

The syntax for a subclass definition looks like this:

```
class DerivedClassName (BaseClassName) :  
    pass
```

Of course, usually we will have an indented block with the class attributes and methods instead of merely a pass statement. The name BaseClassName must be defined in a scope containing the derived class definition. With all this said, we can implement our Person and Employee class:

```
class Person:  
  
    def __init__(self, first, last):  
        self.firstname = first  
        self.lastname = last  
  
    def Name(self):  
        return self.firstname + " " + self.lastname  
  
class Employee(Person):  
  
    def __init__(self, first, last, staffnum):  
        Person.__init__(self, first, last)  
        self.staffnumber = staffnum  
  
    def GetEmployee(self):  
        return self.Name() + ", " + self.staffnumber  
  
x = Person("Marge", "Simpson")  
y = Employee("Homer", "Simpson", "1007")  
  
print(x.Name())  
print(y.GetEmployee())
```

Our program returns the following output:

```
$ python3 person.py  
Marge Simpson  
Homer Simpson, 1007
```

The `__init__` method of our `Employee` class explicitly invokes the `__init__` method of the `Person` class. We could have used `super` instead. `super().__init__(first, last)` is automatically replaced by a call to the superclasses method, in this case `__init__`:

```
def __init__(self, first, last, staffnum):
    super().__init__(first, last)
    self.staffnumber = staffnum
```

Please note that we used `super()` without arguments. This is only possible in Python3. We could have written "`super(Employee, self).__init__(first, last, age)`" which still works in Python3 and is compatible with Python2.

OVERLOADING AND OVERRIDING

Instead of using the methods "Name" and "GetEmployee" in our previous example, it might have been better to put this functionality into the "`__str__`" method. In doing so, we gain a lot, especially a leaner design. We have a string casting for our classes and we can simply print out instances. Let's start with a `__str__` method in `Person`:

```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first, last, staffnum):
        super().__init__(first, last)
        self.staffnumber = staffnum

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x)
print(y)
```

The output looks like this:

```
$ python3 person2.py
Marge Simpson
Homer Simpson
```

First of all, we can see that if we print an instance of the `Employee` class, the `__str__` method of `Person` is used. This is due to inheritance. The only problem we have now is the

fact that the output of "print(y)" is not the same as the "print(y.GetEmployee())". This means that our Employee class needs its own `__str__` method. We could write it like this:

```
def __str__(self):
    return self.firstname + " " + self.lastname + ", " +
    self.staffnumber
```

But it is a lot better to use the `__str__` method of Person inside of the new definition. This way, we can make sure that the output of the Employee `__str__` method will automatically change, if the `__str__` method from the superclass Person changes. We could, for example, add a new attribute age in Person:

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " +
        str(self.age)

class Employee(Person):

    def __init__(self, first, last, age, staffnum):
        super().__init__(first, last, age)
        self.staffnumber = staffnum

    def __str__(self):
        return super().__str__() + ", " + self.staffnumber

x = Person("Marge", "Simpson", 36)
y = Employee("Homer", "Simpson", 28, "1007")

print(x)
print(y)
```

We have overridden the method `__str__` from Person in Employee. By the way, we have overridden `__init__` also. Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its superclass or by one of its superclasses. The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.

Overwriting is not a different concept but usually a term wrongly used for overriding!

In the context of object-oriented programming, you might have heard about "overloading" as well. Overloading is the ability to define the same method, with the same name but with a different number of arguments and types. It's the ability of one function to perform

different tasks, depending on the number of parameters or the types of the parameters.

Let's look first at the case, in which we have the same number of parameters but different types for the parameters. When we define a function in Python, we don't have to and we can't declare the types of the parameters. So if we define the function "successor" in the following example, we implicitly define a family of function, i.e. a function, which can work on integer values, one which can cope with float values and so. Of course, there are types which will lead to an error if used:

```
>>> def successor(number):
...     return number + 1
...
>>> successor(1)
2
>>> successor(1.6)
2.6
>>> successor([3,5,9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in successor
TypeError: can only concatenate list (not "int") to list
>>>
```

You can skip the following paragraphs with the comparisons with C++, if you want to.

This course is not about C++ and we have so far avoided using any C++ code. We want to make an exception now, so that you can see, how overloading works in C++. While we had just one definition in Python, we have two function definitions in C++, i.e. one for the type "int" and one for "double":

```
#include <iostream>
#include <cstdlib>

using namespace std;

int successor(int number) {
    return number + 1;
}

double successor(double number) {
    return number + 1;
}

int main() {

    cout << successor(10) << endl;
    cout << successor(10.3) << endl;
```

```
    return 0;  
}
```

Having a function with a different number of parameters is another way of function overloading. The following C++ program shows such an example. The function f can be called with either one or two integer arguments:

```
#include <iostream>  
using namespace std;  
  
int f(int n);  
int f(int n, int m);  
  
int main() {  
  
    cout << "f(3): " << f(3) << endl;  
    cout << "f(3, 4): " << f(3, 4) << endl;  
    return 0;  
}  
  
int f(int n) {  
    return n + 42;  
}  
int f(int n, int m) {  
    return n + m + 42;  
}
```

This doesn't work in Python, as we can see in the following example. The second definition of f with two parameters redefines or overrides the first definition with one argument. Overriding means that the first definition is not available anymore. This explains the error message:

```
>>> def f(n):  
...     return n + 42  
...  
>>> def f(n,m):  
...     return n + m + 42  
...  
>>> f(3,4)  
49  
>>> f(3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: f() takes exactly 2 arguments (1 given)  
>>>
```

If we need such a behaviour, we can simulate it with default parameters:

```
def f(n, m=None):
    if m:
        return n + m + 42
    else:
        return n + 42
```

The * operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters:

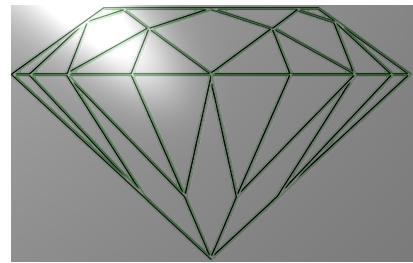
```
def f(*x):
    if len(x) == 1:
        return x[0] + 42
    else:
        return x[0] + x[1] + 42
```


MULTIPLE INHERITANCE

INTRODUCTION

In the previous chapter of our tutorial, we have covered inheritance, or more specific "single inheritance". As we have seen, a class inherits in this case from one class.

Multiple inheritance on the other hand is a feature in which a class can inherit attributes and methods from more than one parent class. The critics point out that multiple inheritance comes along with a high level of complexity and ambiguity in situations such as the diamond problem. We will address this problem later in this chapter.

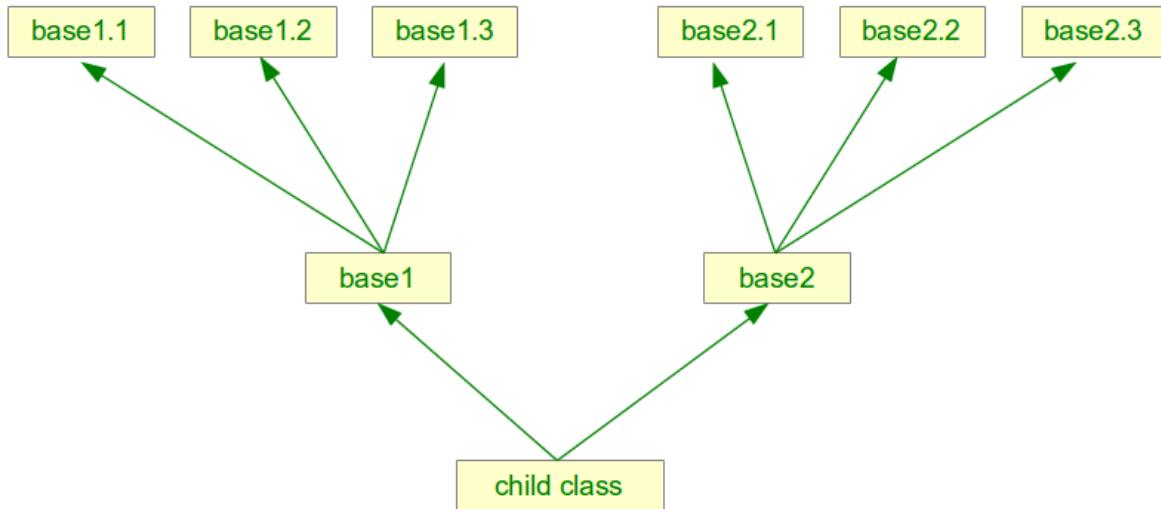


The widespread prejudice that multiple inheritance is something "dangerous" or "bad" is mostly nourished by programming languages with poorly implemented multiple inheritance mechanisms and above all by improper usage of it. Java doesn't even support multiple inheritance, while C++ supports it. Python has a sophisticated and well-designed approach to multiple inheritance.

A class definition, where a child class SubClassName inherits from the parent classes BaseClass1, BaseClass2, BaseClass3, and so on, looks like this:

```
class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):  
    pass
```

It's clear that all the superclasses BaseClass1, BaseClass2, BaseClass3, ... can inherit from other superclasses as well. What we get is an inheritance tree.



EXAMPLE: CALENDARCLOCK

We want to introduce the principles of multiple inheritance with an example. For this purpose, we will implement two independent classes: a "Clock" and a "Calendar" class. After this, we will introduce a class "CalendarClock", which is, as the name implies, a combination of "Clock" and "Calendar". CalendarClock inherits both from "Clock" and "Calendar".



The class Clock simulates the tick-tack of a clock. An instance of this class contains the

time, which is stored in the attributes `self.hours`, `self.minutes` and `self.seconds`. Principally, we could have written the `__init__` method and the set method like this:

```
def __init__(self, hours=0, minutes=0, seconds=0):
    self._hours = hours
    self._minutes = minutes
    self._seconds = seconds

def set(self, hours, minutes, seconds=0):
    self._hours = hours
    self._minutes = minutes
    self._seconds = seconds
```

We decided against this implementation, because we added additional code for checking the plausibility of the time data into the set method. We call the set method from the `__init__` method as well, because we want to circumvent redundant code.

The complete Clock class:

```
"""
The class Clock is used to simulate a clock.
"""

class Clock(object):

    def __init__(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        self.set_clock(hours, minutes, seconds)

    def set_clock(self, hours, minutes, seconds):
        """
        The parameters hours, minutes and seconds have to be
        integers and must satisfy the following equations:
        0 <= h < 24
        0 <= m < 60
        0 <= s < 60
        """

        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and
23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self._minutes = minutes
        else:
```

```

        raise TypeError("Minutes have to be integers between 0
and 59!")
    if type(seconds) == int and 0 <= seconds and seconds < 60:
        self.__seconds = seconds
    else:
        raise TypeError("Seconds have to be integers between 0
and 59!")

def __str__(self):
    return "{0:02d}:{1:02d}:{2:02d}".format(self._hours,
                                              self._minutes,
                                              self._seconds)

def tick(self):
    """
    This method lets the clock "tick", this means that the
    internal time will be advanced by one second.

    Examples:
    >>> x = Clock(12,59,59)
    >>> print(x)
    12:59:59
    >>> x.tick()
    >>> print(x)
    13:00:00
    >>> x.tick()
    >>> print(x)
    13:00:01
    """

    if self.__seconds == 59:
        self.__seconds = 0
        if self.__minutes == 59:
            self.__minutes = 0
            if self._hours == 23:
                self._hours = 0
            else:
                self._hours += 1
        else:
            self.__minutes += 1
    else:
        self.__seconds += 1

if __name__ == "__main__":
    x = Clock(23,59,59)
    print(x)
    x.tick()
    print(x)
    y = str(x)
    print(type(y))

```

If you call this module standalone, you get the following output:

```
$ python3 clock.py
23:59:59
00:00:00
<class 'str'>
```

Let's check our exception handling by inputting floats and strings as input. We also check, what happens, if we exceed the limits of the expected values:

```
>>> from clock import Clock
>>> x = Clock(7.7,45,17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "clock.py", line 16, in __init__
    self.set_Clock(hours, minutes, seconds)
  File "clock.py", line 30, in set_Clock
    raise TypeError("Hours have to be integers between 0 and 23!")
TypeError: Hours have to be integers between 0 and 23!
>>> x = Clock(24,45,17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "clock.py", line 16, in __init__
    self.set_Clock(hours, minutes, seconds)
  File "clock.py", line 30, in set_Clock
    raise TypeError("Hours have to be integers between 0 and 23!")
TypeError: Hours have to be integers between 0 and 23!
>>> x = Clock(23,60,17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "clock.py", line 16, in __init__
    self.set_Clock(hours, minutes, seconds)
  File "clock.py", line 34, in set_Clock
    raise TypeError("Minutes have to be integers between 0 and 59!")
TypeError: Minutes have to be integers between 0 and 59!
>>> x = Clock("23","60","17")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "clock.py", line 16, in __init__
    self.set_Clock(hours, minutes, seconds)
  File "clock.py", line 30, in set_Clock
    raise TypeError("Hours have to be integers between 0 and 23!")
TypeError: Hours have to be integers between 0 and 23!
>>>
>>> x = Clock(23,17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 4 arguments (3 given)
>>>
```

We will now create a class "Calendar", which has lots of similarities to the previously defined Clock class. Instead of "tick" we have an "advance" method, which advances the date by one day, whenever it is called. Adding a day to a date is quite tricky. We have to

check, if the date is the last day in a month and the number of days in the months vary. As if this isn't bad enough, we have February and the leap year problem.

The rules for calculating a leap year are the following:

- If a year is divisible by 400, it is a leap year.
- If a year is not divisible by 400 but by 100, it is not a leap year.
- A year number which is divisible by 4 but not by 100, it is a leap year.
- All other year numbers are common years, i.e. no leap years.

As a little useful gimmick, we added a possibility to output a date either in British or in American (Canadian) style.

```
"""
The class Calendar implements a calendar.
"""

class Calendar(object):

    months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
    date_style = "British"

    @staticmethod
    def leapyear(year):
        """
        The method leapyear returns True if the parameter year
        is a leap year, False otherwise
        """
        if not year % 4 == 0:
            return False
        elif not year % 100 == 0:
            return True
        elif not year % 400 == 0:
            return False
        else:
            return True

    def __init__(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """

        self.set_Calendar(d,m,y)

    def set_Calendar(self, d, m, y):
        """
        d, m, y have to be integer values and year has to be
        a four digit year number
        """
```

```

        if type(d) == int and type(m) == int and type(y) == int:
            self.__days = d
            self.__months = m
            self.__years = y
        else:
            raise TypeError("d, m, y have to be integers!")

    def __str__(self):
        if Calendar.date_style == "British":
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__days,
                                                    self.__months,
                                                    self.__years)
        else:
            # assuming American style
            return "{0:02d}/{1:02d}/{2:4d}".format(self.__months,
                                                    self.__days,
                                                    self.__years)

    def advance(self):
        """
        This method advances to the next date.
        """

        max_days = Calendar.months[self.__months-1]
        if self.__months == 2 and Calendar.leapyear(self.__years):
            max_days += 1
        if self.__days == max_days:
            self.__days = 1
            if self.__months == 12:
                self.__months = 1
                self.__years += 1
            else:
                self.__months += 1
        else:
            self.__days += 1

    if __name__ == "__main__":
        x = Calendar(31,12,2012)
        print(x, end=" ")
        x.advance()
        print("after applying advance: ", x)
        print("2012 was a leapyear:")
        x = Calendar(28,2,2012)
        print(x, end=" ")
        x.advance()
        print("after applying advance: ", x)
        x = Calendar(28,2,2013)
        print(x, end=" ")
        x.advance()
        print("after applying advance: ", x)
        print("1900 no leapyear: number divisible by 100 but not by 400:")
    )
    x = Calendar(28,2,1900)

```

```

print(x, end=" ")
x.advance()
print("after applying advance: ", x)
print("2000 was a leapyear, because number divisible by 400: ")
x = Calendar(28,2,2000)
print(x, end=" ")
x.advance()
print("after applying advance: ", x)
print("Switching to American date style: ")
Calendar.date_style = "American"
print("after applying advance: ", x)

```

Starting this script provides us with the following results:

```

$ python3 calendar.py
31.12.2012 after applying advance: 01.01.2013
2012 was a leapyear:
28.02.2012 after applying advance: 29.02.2012
28.02.2013 after applying advance: 01.03.2013
1900 no leapyear: number divisible by 100 but not by 400:
28.02.1900 after applying advance: 01.03.1900
2000 was a leapyear, because number divisible by 400:
28.02.2000 after applying advance: 29.02.2000
Switching to American date style:
after applying advance: 02/29/2000

```

At last, we will come to our multiple inheritance example. We are now capable of implementing the originally intended class `CalendarClock`, which will inherit from both `Clock` and `Calendar`. The method "tick" of `Clock` will have to be overridden. However, the new tick method of `CalendarClock` has to call the tick method of `Clock`: `Clock.tick(self)`

```

"""
Modul, which implements the class CalendarClock.
"""

from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):
    """
        The class CalendarClock implements a clock with integrated
        calendar. It's a case of multiple inheritance, as it
inherits
        both from Clock and Calendar
    """

    def __init__(self, day, month, year, hour, minute, second):
        Clock.__init__(self, hour, minute, second)

```

```

        Calendar.__init__(self, day, month, year)

    def tick(self):
        """
        advance the clock by one second
        """
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)

if __name__ == "__main__":
    x = CalendarClock(31, 12, 2013, 23, 59, 59)
    print("One tick from ", x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(28, 2, 1900, 23, 59, 59)
    print("One tick from ", x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(28, 2, 2000, 23, 59, 59)
    print("One tick from ", x, end=" ")
    x.tick()
    print("to ", x)

    x = CalendarClock(7, 2, 2013, 13, 55, 40)
    print("One tick from ", x, end=" ")
    x.tick()
    print("to ", x)

```

The output of the program hopefully further clarifies what's going on in this class:

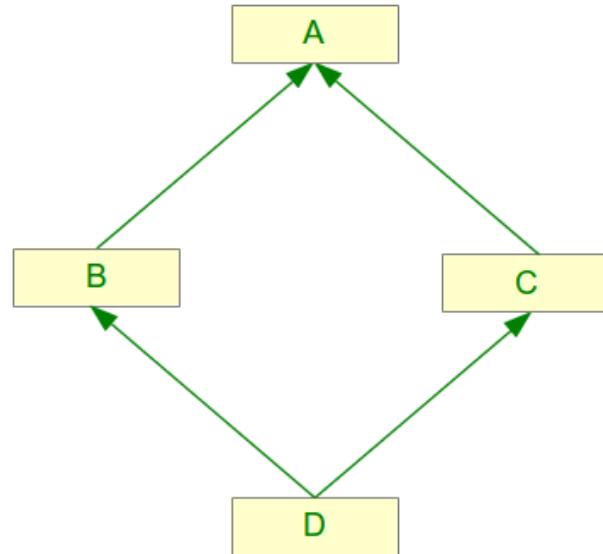
```
$ python3 calendar_clock.py
One tick from 31/12/2013, 23:59:59 to 01/01/2014, 00:00:00
One tick from 28/02/1900, 23:59:59 to 01/03/1900, 00:00:00
One tick from 28/02/2000, 23:59:59 to 29/02/2000, 00:00:00
One tick from 07/02/2013, 13:55:40 to 07/02/2013, 13:55:41
```

THE DIAMOND PROBLEM OR THE „DEADLY DIAMOND OF DEATH“

The "diamond problem" (sometimes referred to as the "deadly diamond of death") is the generally used term for an ambiguity that arises when two classes B and C inherit from a

superclass A, and another class D inherits from both B and C. If there is a method "m" in A that B or C (or even both of them) has overridden, and furthermore, if does not override this method, then the question is which version of the method does D inherit? It could be the one from A, B or C

Let's look at Python. The first Diamond Problem configuration is like this: Both B and C override the method m of A:



```

class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
  
```

If you call the method m on an instance x of D, i.e. x.m(), we will get the output "m of B called". If we transpose the order of the classes in the class header of D in "class D(C,B):", we will get the output "m of C called".

The case in which m will be overridden only in one of the classes B or C, e.g. in C:

```

class A:
    def m(self):
        print("m of A called")

class B(A):
    pass

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass

x = D()
x.m()
  
```

Principally, two possibilities are imaginable: "m of C" or "m of A" could be used

We call this script with Python2.7 (python) and with Python3 (python3) to see what's happening:

```
$ python diamond1.py
m of A called
$ python3 diamond1.py
m of C called
```

Only for those who are interested in Python version2:

To have the same inheritance behaviour in Python2 as in Python3, every class has to inherit from the class "object". Our class A doesn't inherit from object, so we get a so-called old-style class, if we call the script with python2. Multiple inheritance with old-style classes is governed by two rules: depth-first and then left-to-right. If you change the header line of A into "class A(object):", we will have the same behaviour in both Python versions.

SUPER AND MRO

We have seen in our previous implementation of the diamond problem, how Python "solves" the problem, i.e. in which order the base classes are browsed through. The order is defined by the so-called "Method Resolution Order" or in short MRO.^{1\}

We will extend our previous example, so that every class defines its own method m:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    def m(self):
        print("m of D called")
```

Let's apply the method m on an instance of D. We can see that only the code of the method m of D will be executed. We can also explicitly call the methods m of the other classes via

the class name, as we demonstrate in the following interactive Python session:

```
>>> from super1 import A,B,C,D
>>> x = D()
>>> B.m(x)
m of B called
>>> C.m(x)
m of C called
>>> A.m(x)
m of A called
```

Now let's assume that the method m of D should execute the code of m of B, C and A as well, when it is called. We could implement it like this:

```
class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
        A.m(self)
```

The output is what we have been looking for:

```
>>> from mro import D
>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

But it turns out once more that things are more complicated than it seems. How can we cope with the situation, if both m of B and m of C will have to call m of A as well. In this case, we have to take away the call A.m(self) from m in D. The code might look like this, but there is still a bug lurking in it:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        A.m(self)

class C(A):
    def m(self):
```

```

        print("m of C called")
        A.m(self)

class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)

```

The bug is that the method m of A will be called twice:

```

>>> from super3 import D
>>> x = D()
>>> x.m()
m of D called
m of B called
m of A called
m of C called
m of A called

```

One way to solve this problem - admittedly not a Pythonic one - consists in splitting the methods m of B and C in two methods. The first method, called `_m` consists of the specific code for B and C and the other method is still called m, but consists now of a call "`self._m()`" and a call "`A.m(self)`". The code of the method m of D consists now of the specific code of D '`print("m of D called")`', and the calls `B._m(self)`, `C._m(self)` and `A.m(self)`:

```

class A:
    def m(self):
        print("m of A called")

class B(A):
    def _m(self):
        print("m of B called")
    def m(self):
        self._m()
        A.m(self)

class C(A):
    def _m(self):
        print("m of C called")
    def m(self):
        self._m()
        A.m(self)

class D(B,C):
    def m(self):
        print("m of D called")
        B._m(self)
        C._m(self)
        A.m(self)

```

Our problem is solved, but - as we have already mentioned - not in a pythonic way:

```
>>> from super4 import D
>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

The optimal way to solve the problem, which is the "super" pythonic way, consists in calling the super function:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        super().m()

class C(A):
    def m(self):
        print("m of C called")
        super().m()

class D(B,C):
    def m(self):
        print("m of D called")
        super().m()
```

It also solves our problem, but in a beautiful design as well:

```
>>> from super5 import D
>>> x = D()
>>> x.m()
m of D called
m of B called
m of C called
m of A called
```

The super function is often used when instances are initialized with the `__init__` method:

```

class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()

class D(B,C):
    def __init__(self):
        print("D.__init__")
        super().__init__()

```

We demonstrate the way of working in the following interactive session:

```

>>> from super_init import A,B,C,D
>>> d = D()
D.__init__
B.__init__
C.__init__
A.__init__
>>> c = C()
C.__init__
A.__init__
>>> b = B()
B.__init__
A.__init__
>>> a = A()
A.__init__

```

The question arises how the super functions makes its decision. How does it decide which class has to be used? As we have already mentioned, it uses the so-called method resolution order(MRO). It is based on the "C3 superclass linearisation" algorithm. This is called a linearisation, because the tree structure is broken down into a linear order. The mro method can be used to create this list:

```

>>> from super_init import A,B,C,D
>>> D.mro()
[<class 'super_init.D'>, <class 'super_init.B'>, <class
'super_init.C'>, <class 'super_init.A'>, <class 'object'>]
>>> B.mro()
[<class 'super_init.B'>, <class 'super_init.A'>, <class 'object'>]
>>> A.mro()
[<class 'super_init.A'>, <class 'object'>]

```

POLYMORPHISM

Polymorphism is construed from two Greek words. "Poly" stands for "much" or "many" and "morph" means shape or form.

Polymorphism is the state or condition of being polymorphous, or if we use the translations of the components "the ability to be in many shapes or forms.

Polymorphism is a term used in many scientific areas. In Crystallography it defines the state, if something crystallizes into two or more chemically identical but crystallographically distinct forms.

Biologists know polymorphism as the existence of an organism in several form or colour varieties. The Romans even had a god, called Morpheus, who is able to take any human form: Morpheus appears in Ovid's metamorphoses and is the son of Somnus, the god of sleep. You can admire Morpheus and Iris in the picture on the right side.



So, before we fall to sleep, we get back to Python and to what polymorphism means in the programming language context. Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms. We can have in some programming languages polymorphic functions or methods, for example. Polymorphic functions or methods can be applied to arguments of different types, and they can behave differently depending on the type of the arguments to which they are applied. We can also define the same function name with a varying number of parameters.

Let's have a look at the following Python function:

```
def f(x, y):  
    print("values: ", x, y)
```

```
f(42, 43)
f(42, 43.7)
f(42.3, 43)
f(42.0, 43.9)
```

We can call this function with various types, as demonstrated in the example. In typed programming languages like Java or C++, we would have to overload f to implement the various type combinations.

Our example could be implemented like this in C++:

```
#include <iostream>
using namespace std;

void f(int x, int y) {
    cout << "values: " << x << ", " << y << endl;
}

void f(int x, double y) {
    cout << "values: " << x << ", " << y << endl;
}

void f(double x, int y) {
    cout << "values: " << x << ", " << y << endl;
}

void f(double x, double y) {
    cout << "values: " << x << ", " << y << endl;
}

int main()
{
    f(42, 43);
    f(42, 43.7);
    f(42.3, 43);
    f(42.0, 43.9);
}
```

Python is implicitly polymorphic. We can apply our previously defined function f even to lists, strings or other types, which can be printed:

```
>>> def f(x,y):
...     print("values: ", x, y)
...
>>> f([3,5,6],(3,5))
values: [3, 5, 6] (3, 5)
>>> f("A String", ("A tuple", "with Strings"))
values: A String ('A tuple', 'with Strings')
```

```
>>> f({2,3,9}, {"a":3.4,"b":7.8, "c":9.04})  
values: {9, 2, 3} {'a': 3.4, 'c': 9.04, 'b': 7.8}  
>>>
```

FOOTNOTES

1

Python has used since 2.3 the „C3 superclass linearisation“-algorithm to determine the MRO. -->

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein

MAGIC METHODS AND OPERATOR OVERLOADING

INTRODUCTION

The so-called magic methods have nothing to do with wizardry. You have already seen them in previous chapters of our tutorial. They are special methods with fixed names. They are the methods with this clumsy syntax, i.e. the double underscores at the beginning and the end. They are also hard to talk about. How do you pronounce or say a method name like `__init__`? "Underscore underscore init underscore underscore" sounds horrible and is nearly a tongue twister. "Double underscore init double underscore" is a lot better, but the ideal way is "dunder init dunder"¹ That's why magic methods are sometimes called dunder methods!



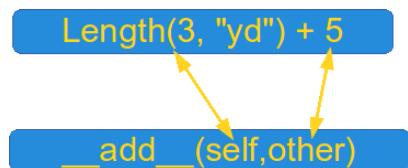
So what's magic about the `__init__` method? The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance `x` of a class `A` with the statement "`x = A()`", Python will do the necessary calls to `__new__` and `__init__`.

Nearly at the end of this chapter of our tutorial we will introduce the `__call__` method. It is overlooked by many beginners and even advanced programmers of Python. It is a functionality which many programming languages do not have, so programmers are generally not looking for. The `__call__` method enables Python programmers to write classes where the instances behave like functions. Both functions and the instances of such classes are called callables.

We have encountered the concept of operator overloading many times in the course of this tutorial. We had used the plus sign to add numerical values, to concatenate strings or to combine lists:

```
>>> 4 + 5
9
>>> 3.8 + 9
12.8
>>> "Peter" + " " + "Pan"
'Peter Pan'
>>> [3,6,8] + [7,11,13]
[3, 6, 8, 7, 11, 13]
>>>
```

It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the `__add__` method. For "-" it is "`__sub__`" and so on. We have a complete listing of all the magic methods a little bit further down.



The mechanism works like this: If we have an expression "`x + y`" and `x` is an instance of class `K`, then Python will check the class definition of `K`. If `K` has a method `__add__` it will be called with `x.__add__(y)`, otherwise we will get an error message.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'K' and 'K'
```

OVERVIEW OF MAGIC METHODS

BINARY OPERATORS

Operator	Method
+	<code>object.__add__(self, other)</code>
-	<code>object.__sub__(self, other)</code>
*	<code>object.__mul__(self, other)</code>
//	<code>object.__floordiv__(self, other)</code>
/	<code>object.__truediv__(self, other)</code>
%	<code>object.__mod__(self, other)</code>
**	<code>object.__pow__(self, other[, modulo])</code>
<<	<code>object.__lshift__(self, other)</code>

```
>> object.__rshift__(self, other)
& object.__and__(self, other)
^ object.__xor__(self, other)
| object.__or__(self, other)
```

EXTENDED ASSIGNMENTS

Operator	Method
<code>+=</code>	<code>object.__iadd__(self, other)</code>
<code>-=</code>	<code>object.__isub__(self, other)</code>
<code>*=</code>	<code>object.__imul__(self, other)</code>
<code>/=</code>	<code>object.__idiv__(self, other)</code>
<code>//=</code>	<code>object.__ifloordiv__(self, other)</code>
<code>%=</code>	<code>object.__imod__(self, other)</code>
<code>**=</code>	<code>object.__ipow__(self, other[, modulo])</code>
<code><<=</code>	<code>object.__ilshift__(self, other)</code>
<code>>>=</code>	<code>object.__irshift__(self, other)</code>
<code>&=</code>	<code>object.__iand__(self, other)</code>
<code>^=</code>	<code>object.__ixor__(self, other)</code>
<code> =</code>	<code>object.__ior__(self, other)</code>

UNARY OPERATORS

Operator	Method
<code>-</code>	<code>object.__neg__(self)</code>
<code>+</code>	<code>object.__pos__(self)</code>
<code>abs()</code>	<code>object.__abs__(self)</code>
<code>~</code>	<code>object.__invert__(self)</code>
<code>complex()</code>	<code>object.__complex__(self)</code>
<code>int()</code>	<code>object.__int__(self)</code>
<code>long()</code>	<code>object.__long__(self)</code>
<code>float()</code>	<code>object.__float__(self)</code>
<code>oct()</code>	<code>object.__oct__(self)</code>
<code>hex()</code>	<code>object.__hex__(self</code>

COMPARISON OPERATORS

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

EXAMPLE CLASS: LENGTH

We will demonstrate in the following Length class, how you can overload the "+" operator for your own class. To do this, we have to overload the `__add__` method. Our class contains the `__str__` and `__repr__` methods as well. The instances of the class Length contain length or distance information. The attributes of an instance are `self.value` and `self.unit`.

This class allows us to calculate expressions with mixed units like this one:

2.56 m + 3 yd + 7.8 in + 7.03 cm

The class can be used like this:

```
>>> from unit_conversions import Length
>>> L = Length
>>> print(L(2.56,"m") + L(3,"yd") + L(7.8,"in") + L(7.03,"cm"))
5.57162
>>>
```

The listing of the class:

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
               "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
               "mi" : 1609.344 }

    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit
```

```

def Converse2Metres(self):
    return self.value * Length.__metric[self.unit]

def __add__(self, other):
    l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)

def __str__(self):
    return str(self.Converse2Metres())

def __repr__(self):
    return "Length(" + str(self.value) + ", '" + self.unit +
")"

if __name__ == "__main__":
    x = Length(4)
    print(x)
    y = eval(repr(x))

    z = Length(4.5, "yd") + Length(1)
    print(repr(z))
    print(z)

```

If we start this program, we get the following output:

```

4
Length(5.593613298337708, 'yd')
5.1148

```

We use the method `__iadd__` to implement the extended assignment:

```

def __iadd__(self, other):
    l = self.Converse2Metres() + other.Converse2Metres()
    self.value = l / Length.__metric[self.unit]
    return self

```

Now we are capable to write the following assignments:

```

x += Length(1)
x += Length(4, "yd")

```

We have added 1 metre in the example above by writing "`x += Length(1)`". Most certainly, you will agree with us that it would be more convenient to simply write "`x += 1`" instead. We also want to treat expressions like "`Length(5,"yd") + 4.8`" similarly. So, if somebody

uses a type int or float, our class takes it automatically for "metre" and converts it into a Length object. It's easy to adapt our `__add__` and "`__iadd__`" method for this task. All we have to do is to check the type of the parameter "other":

```
def __add__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)

def __iadd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    self.value = l / Length.__metric[self.unit]
    return self
```

It's a safe bet that if somebody works for a while with adding integers and floats from the right sight that he or she wants to the same from the left side! So let's try it out:

```
>>> from unit_conversions import Length
>>> x = Length(3, "yd") + 5
>>> x = 5 + Length(3, "yd")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Length'
>>>
```

Of course, the left side has to be of type "Length", because otherwise Python tries to apply the `__add__` method from int, which can't cope with Length objects as second arguments!

Python provides a solution for this problem as well. It's the `__radd__` method. It works like this: Python tries to evaluate the expression "5 + Length(3, 'yd')". First it calls `int.__add__(5, Length(3, 'yd'))`, which will raise an exception. After this it will try to invoke `Length.__radd__(Length(3, "yd"), 5)`. It's easy to recognize that the implementation of `__radd__` is analogue to `__add__`:

```
def __radd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() +
otherLength.__radd__(Length(3, "yd"), 5)
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)
```

It's advisable to make use of the `__add__` method in the `__radd__` method:

```
def __radd__(self, other):
    return Length.__add__(self,other)
```

The following diagram illustrates the relationship between `__add__` and `__radd__`:



THE `__CALL__` METHOD

The `__call__` method can be used to turn the instances of the class into callables. Functions are callable objects. A callable object is an object which can be used and behaves like a function but might not be a function. By using the `__call__` method it is possible to define classes in a way that the instances will be callable objects. The `__call__` method is called, if the instance is called "like a function", i.e. using brackets. The following example defines a class with which we can create arbitrary polynomial functions:

```
class Polynomial:

    def __init__(self, *coefficients):
        self.coefficients = coefficients[::-1]

    def __call__(self, x):
        res = 0
        for index, coeff in enumerate(self.coefficients):
            res += coeff * x** index
        return res

    # a constant function
p1 = Polynomial(42)

    # a straight Line
p2 = Polynomial(0.75, 2)

    # a third degree Polynomial
p3 = Polynomial(1, -0.5, 0.75, 2)

for i in range(1, 10):
    print(i, p1(i), p2(i), p3(i))
```

These are the results of the previous function:

```

1 42 2.75 3.25
2 42 3.5 9.5
3 42 4.25 26.75
4 42 5.0 61.0
5 42 5.75 118.25
6 42 6.5 204.5
7 42 7.25 325.75
8 42 8.0 488.0
9 42 8.75 697.25

```

You will find further interesting examples of the `__call__` function in our tutorial in the chapters **Decorators** and **Memoization with Decorators**. You may also consult our chapter on **Polynomials**

STANDARD CLASSES AS BASE CLASSES

It's possible to use standard classes - like `int`, `float`, `dict` or `lists` - as base classes as well.

We extend the `list` class by adding a `push` method:

```

class Plist(list):

    def __init__(self, l):
        list.__init__(self, l)

    def push(self, item):
        self.append(item)

if __name__ == "__main__":
    x = Plist([3,4])
    x.push(47)
    print(x)

```

This means that all the previously introduced binary and extended assignment operators exist in the "reversed" version as well:

```

__radd__
__rsub__
__rmul__
...
and so on

```

EXERCISES

1. Write a class with the name Ccy, similar to the previously defined Length class.

Ccy should contain values in various currencies, e.g. "EUR", "GBP" or "USD". An instance should contain the amount and the currency unit.

The class, you are going to design as an excercise, might be best described with the following example session:



```
>>> from currencies import Ccy
>>> v1 = Ccy(23.43, "EUR")
>>> v2 = Ccy(19.97, "USD")
>>> print(v1 + v2)
32.89 EUR
>>> print(v2 + v1)
31.07 USD
>>> print(v1 + 3) # an int or a float is considered to be a EUR
value
27.43 EUR
>>> print(3 + v1)
27.43 EUR
>>>
```

SOLUTIONS TO OUR EXERCISES

1. First exercise:

```
"""
```

The class "Ccy" can be used to define money values in various currencies. A Ccy instance has the string attributes 'unit' (e.g. 'CHF', 'CAD' od 'EUR' and the 'value' as a float.

A currency object consists of a value and the corresponding unit.

```
"""
class Ccy:

    currencies = {'CHF': 1.0821202355817312,
                  'CAD': 1.488609845538393,
                  'GBP': 0.8916546282920325,
                  'JPY': 114.38826536281809,
                  'EUR': 1.0,
                  'USD': 1.11123458162018}

    def __init__(self, value, unit="EUR"):
        self.value = value
        self.unit = unit

    def __str__(self):
        return "{0:5.2f}".format(self.value) + " " + self.unit

    def changeTo(self, new_unit):
        """
        An Ccy object is transformed from the unit "self.unit"
        to "new_unit"
        """
        self.value = (self.value / Ccy.currencies[self.unit] *
Ccy.currencies[new_unit])
        self.unit = new_unit

    def __add__(self, other):
        """
        Defines the '+' operator.
        If other is a CCy object the currency values
        are added and the result will be the unit of
        self. If other is an int or a float, other will
        be treated as a Euro value.
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
        return Ccy(x + self.value, self.unit)

    def __iadd__(self, other):
        """
        Similar to __add__
        """
        if type(other) == int or type(other) == float:
            x = (other * Ccy.currencies[self.unit])
        else:
            x = (other.value / Ccy.currencies[other.unit] *
```

```

Ccy.currencies[self.unit])
    self.value += x
    return self

def __radd__(self, other):
    res = self + other
    if self.unit != "EUR":
        res.changeTo("EUR")
    return res

# __sub__, __isub__ and __rsub__ can be defined analogue

x = Ccy(10, "USD")
y = Ccy(11)
z = Ccy(12.34, "JPY")
z = 7.8 + x + y + 255 + z
print(z)

lst = [Ccy(10, "USD"), Ccy(11), Ccy(12.34, "JPY"), Ccy(12.34,
"CAD")]

z = sum(lst)

print(z)

```

The program returns:

```

282.91 EUR
28.40 EUR

```

Another interesting aspect of this currency converter class in Python can be shown, if we add multiplication. You will easily understand that it makes no sense to allow expressions like "12.4 € * 3.4" (*or in prefix notation* ":" €12.4* 3.4"), but it makes perfect sense to evaluate "3 * 4.54 €". You can find the new currency converter class with the newly added methods for `__mul__`, `__imul__` and `__rmul__` in the following listing:

```
"""

```

The class "Ccy" can be used to define money values in various currencies. A Ccy instance has the string attributes 'unit' (e.g. 'CHF', 'CAD' or 'EUR' and the 'value' as a float. A currency object consists of a value and the corresponding unit.

```
"""

```

```

class Ccy:

    currencies = {'CHF': 1.0821202355817312,
                  'CAD': 1.488609845538393,

```

```

'GBP': 0.8916546282920325,
'JPY': 114.38826536281809,
'EUR': 1.0,
'USD': 1.11123458162018}

def __init__(self, value, unit="EUR"):
    self.value = value
    self.unit = unit

def __str__(self):
    return "{0:5.2f}".format(self.value) + " " + self.unit

def __repr__(self):
    return 'Ccy(' + str(self.value) + ', "' + self.unit +
')'

def changeTo(self, new_unit):
    """
    An Ccy object is transformed from the unit "self.unit"
    to "new_unit"
    """
    self.value = (self.value / Ccy.currencies[self.unit] *
Ccy.currencies[new_unit])
    self.unit = new_unit

def __add__(self, other):
    """
    Defines the '+' operator.
    If other is a CCy object the currency values
    are added and the result will be the unit of
    self. If other is an int or a float, other will
    be treated as a Euro value.
    """
    if type(other) == int or type(other) == float:
        x = (other * Ccy.currencies[self.unit])
    else:
        x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
    return Ccy(x + self.value, self.unit)

def __iadd__(self, other):
    """
    Similar to __add__
    """
    if type(other) == int or type(other) == float:
        x = (other * Ccy.currencies[self.unit])
    else:
        x = (other.value / Ccy.currencies[other.unit] *
Ccy.currencies[self.unit])
    self.value += x
    return self

def __radd__(self, other):
    res = self + other
    if self.unit != "EUR":
        res.changeTo("EUR")
    return res

```

```
# __sub__, __isub__ and __rsub__ can be defined analogue

def __mul__(self, other):
    """
    Multiplication is only defined as a scalar
    multiplication,
    i.e. a money value can be multiplied by an int or a
    float.
    It is not possible to multiply two money values
    """
    if type(other)==int or type(other)==float:
        return Ccy(self.value * other, self.unit)
    else:
        raise TypeError("unsupported operand type(s) for *:
'Ccy' and " + type(other).__name__)

def __rmul__(self, other):
    return self.__mul__(other)

def __imul__(self, other):
    if type(other)==int or type(other)==float:
        self.value *= other
        return self
    else:
        raise TypeError("unsupported operand type(s) for *:
'Ccy' and " + type(other).__name__)
```

Assuming that you have saved the class under the name currency_converter, you can use it in the following way in the command shell:

```
>>> from currency_converter import Ccy
>>> x = Ccy(10.00, "EUR")
>>> y = Ccy(10.00, "GBP")
>>> x + y
Ccy(21.215104685942173, "EUR")
>>> print(x + y)
21.22 EUR
>>> print(2*x + y*0.9)
30.09 EUR
>>>
```

We can further improve our currency converter class by using a function get_currencies, which downloads the latest exchange rates from finance.yahoo.com. This function returns an exchange rates dictionary in our previously used format. The function is in a module called `exchange_rates.py`. This is the code of the function `exchange_rates.py`:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

def get_currency_rates(base="USD"):
    """
    The file at location url is read in and the exchange
    rates are extracted """

```

```

        url =
"https://finance.yahoo.com/webservice/v1/symbols/allcurrencies/q
uote"
        data = urlopen(url).read()
        data = data.decode('utf-8')
        soup = BeautifulSoup(data, 'html.parser')
        data = soup.get_text()

        flag = False
        currencies = {}
        for line in data.splitlines():
            if flag:
                value = float(line)
                flag = False
                currencies[currency] = value
            if line.startswith("USD/"):
                flag = True
                currency = line[4:7]

            currencies["USD"] = 1 # we must add it, because it's not
included in file
            if base != "USD":
                base_currency_rate = currencies[base]
                for currency in currencies:
                    currencies[currency] /= base_currency_rate

        return currencies
    
```

We can import this function from our module. (You have to save it somewhere in your Python path or the directory where your program runs):

```

from exchange_rates import get_currency_rates

class Ccy:

    currencies = get_currencies()

    # continue with the code from the previous version
    
```

We save this version as currency_converter_web.

```

>>> from currency_converter_web import Ccy
>>> x = Ccy(1000, "JPY")
>>> y = Ccy(10, "CHF")
>>> z = Ccy(15, "CAD")
>>> print(2*x + 4.11*y + z)
7722.98 JPY
>>>
    
```

FOOTNOTES

¹ as suggested by Mark Jackson

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

INHERITANCE EXAMPLE

INTRODUCTION

There aren't many good examples on inheritance available on the web. They are either extremely simple and artificial or they are way to complicated. We want to close the gap by providing an example which is on the one hand more realistic - but still not realistic - and on the other hand simple enough to see and understand the basic aspects of inheritance. In our previous chapter, we introduced inheritance formally.



To this purpose we define two base classes: One is an implementation of a clock and the other one of a calendar. Based on these two classes, we define a class `CalendarClock`, which inherits both from the class `Calendar` and from the class `Clock`.

THE CLOCK CLASS

```
class Clock(object):

    def __init__(self, hours=0, minutes=0, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def set(self, hours, minutes, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def tick(self):
        """ Time will be advanced by one second """
        if self.__seconds == 59:
            self.__seconds = 0
            if (self.__minutes == 59):
                self.__minutes = 0
                self.__hours = 0 if self.__hours==23 else
self.__hours + 1
            else:
                self.__minutes += 1;
        else:
            self.__seconds += 1;
```

```

def display(self):
    print("%d:%d:%d" % (self.__hours, self.__minutes,
self.__seconds))

def __str__(self):
    return "%2d:%2d:%2d" % (self.__hours, self.__minutes,
self.__seconds)

x = Clock()
print(x)
for i in range(10000):
    x.tick()
print(x)

```

THE CALENDAR CLASS

```

class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    def __init__(self, day=1, month=1, year=1900):
        self.__day = day
        self.__month = month
        self.__year = year

    def leapyear(self,y):
        if y % 4:
            # not a leap year
            return 0;
        else:
            if y % 100:
                return 1;
            else:
                if y % 400:
                    return 0
                else:
                    return 1;

    def set(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year

    def get():
        return (self, self.__day, self.__month, self.__year)
    def advance(self):
        months = Calendar.months
        max_days = months[self.__month-1]
        if self.__month == 2:
            max_days += self.leapyear(self.__year)
        if self.__day == max_days:
            self.__day = 1

```

```

        if (self.__month == 12):
            self.__month = 1
            self.__year += 1
        else:
            self.__month += 1
    else:
        self.__day += 1

    def __str__(self):
        return str(self.__day) + "/" + str(self.__month) + "/" +
str(self.__year)

if __name__ == "__main__":
    x = Calendar()
    print(x)
    x.advance()
    print(x)

```

THE CALENDAR-CLOCK CLASS

```

from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):

    def __init__(self, day, month, year, hours=0, minutes=0, seconds=0):
        Calendar.__init__(self, day, month, year)
        Clock.__init__(self, hours, minutes, seconds)

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)

if __name__ == "__main__":
    x = CalendarClock(24, 12, 57)
    print(x)
    for i in range(1000):
        x.tick()
    for i in range(1000):
        x.advance()
    print(x)

```

SLOTS

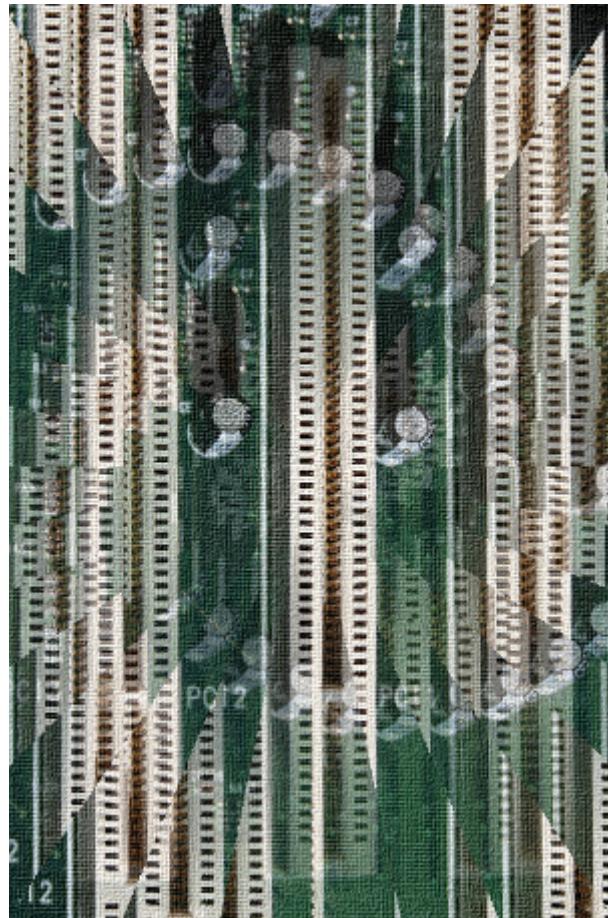
AVOIDING DYNAMICALLY CREATED ATTRIBUTES

The attributes of objects are stored in a dictionary "`__dict__`". Like any other dictionary, a dictionary used for attribute storage doesn't have a fixed number of elements. In other words, you can add elements to dictionaries after they have been defined, as we have seen in our chapter on dictionaries. This is the reason, why you can dynamically add attributes to objects of classes that we have created so far:

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.x = 66
>>> a.y = "dynamically created
attribute"
```

The dictionary containing the attributes of "`a`" can be accessed like this:

```
>>> a.__dict__
{'y': 'dynamically created
attribute', 'x': 66}
```



You might have wondered that you can dynamically add attributes to the classes, we have defined so far, but that you can't do this with built-in classes like '`int`', or '`list`':

```
>>> x = 42
>>> x.a = "not possible to do it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'a'
>>>
>>> lst = [34, 999, 1001]
>>> lst.a = "forget it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'a'
```

Using a dictionary for attribute storage is very convenient, but it can mean a waste of space for objects, which have only a small amount of instance variables. The space consumption can become critical when creating large numbers of instances. Slots are a nice way to work around this space consumption problem. Instead of having a dynamic dict that allows adding attributes to objects dynamically, slots provide a static structure which prohibits additions after the creation of an instance.

When we design a class, we can use slots to prevent the dynamic creation of attributes. To define slots, you have to define a list with the name `__slots__`. The list has to contain all the attributes, you want to use. We demonstrate this in the following class, in which the slots list contains only the name for an attribute "val".

```
class S(object):

    __slots__ = ['val']

    def __init__(self, v):
        self.val = v

x = S(42)
print(x.val)

x.new = "not possible"
```

If we start this program, we can see, that it is not possible to create dynamically a new attribute. We fail to create an attribute "new":

```
42
Traceback (most recent call last):
  File "slots_ex.py", line 12, in <module>
    x.new = "not possible"
AttributeError: 'S' object has no attribute 'new'
```

We mentioned in the beginning that slots are preventing a waste of space with objects. Since Python 3.3 this advantage is not as impressive any more. With Python 3.3 Key-Sharing Dictionaries are used for the storage of objects. The attributes of the instances are capable of sharing part of their internal storage between each other, i.e. the part which stores the keys and their corresponding hashes. This helps to reduce the memory consumption of programs, which create many instances of non-built-in types.

CLASSES AND CLASS CREATION

BEHIND THE SCENES: RELATIONSHIP BETWEEN CLASS AND TYPE

In this chapter of our tutorial, we will provide you with a deeper insight into the magic happening behind the scenes, when we are defining a class or creating an instance of a class. You may ask yourself: "Do I really have to learn these additional details on object oriented programming in Python?" Most probably not, or you belong to the few people who design classes at a very advanced level.



First, we will concentrate on the relationship between type and class. When you have defined classes so far, you may have asked yourself, what is happening "behind the lines". We have already seen, that applying "type" to an object returns the class of which the object is an instance of:

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

The code above returned the following:

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

If you apply type on the name of a class itself, you get the class "type" returned.

```
print(type(list), type(str))
```

The above Python code returned the following:

```
<class 'type'> <class 'type'>
```

This is similar to applying type on type(x) and type(y):

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

This gets us the following result:

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

A user-defined class (or the class "object") is an instance of the class "type". So, we can see, that classes are created from type. In Python3 there is no difference between "classes" and "types". They are in most cases used as synonyms.

The fact that classes are instances of a class "type" allows us to program metaclasses. We can create classes, which inherit from the class "type". So, a metaclass is a subclass of the class "type".

Instead of only one argument, type can be called with three parameters:

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

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

- "classname" is a string defining the class name and becomes the **name** attribute;
- "superclasses" is a list or tuple with the superclasses of our class. This list or tuple will become the **bases** attribute;
- the **attributes_dict** is a dictionary, functioning as the namespace of our class. It contains the definitions for the class body and it becomes the **dict** attribute.

Let's have a look at a simple class definition:

```
class A:
    pass
x = A()
print(type(x))
```

After having executed the Python code above we received the following result:

```
<class '__main__.A'>
```

We can use "type" for the previous class defintion as well:

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

The previous Python code returned the following result:

```
<class '__main__.A'>
```

Generally speaking, this means, that we can define a class A with

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

When we call "type", the **call** method of type is called. The **call** method runs two other methods: **new** and **init**:

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(cls, classname, superclasses, attributedict)
```

The **new** method creates and returns the new class object, and after this the **init** method initializes the newly created object.

```
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__)
```

The previous Python code returned the following result:

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

The class definitions for Robot and Robot2 are syntactically completely different, but they implement logically the same class.

What Python actually does in the first example, i.e. the "usual way" of defining classes, is the following: Python processes the complete class statement from class Robot to collect the methods and attributes of Robot to add them to the attributes_dict of the type call. So, Python will call type in a similar or the same way than we did in Robot2.

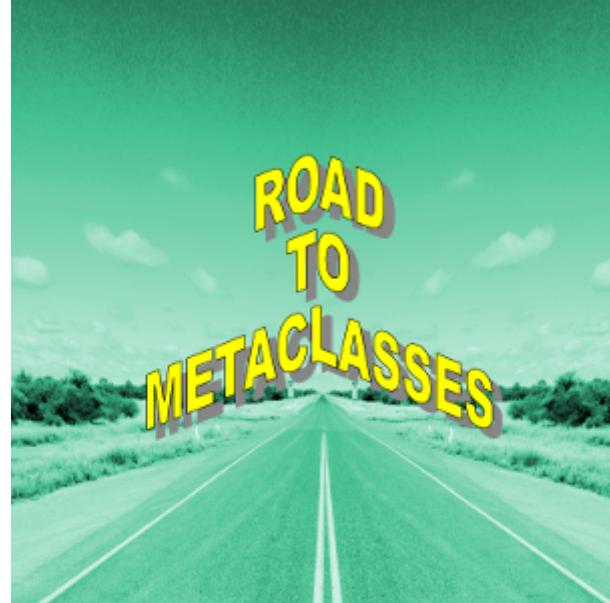
© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

ON THE ROAD TO METACLASSES

MOTIVATION FOR METACLASSES

In this chapter of our tutorial we want to provide some incentives or motivation for the use of metaclasses. To demonstrate some design problems, which can be solved by metaclasses, we will introduce and design a bunch of philosopher classes. Each philosopher class (Philosopher1, Philosopher2, and so on) need the same "set" of methods (in our example just one, i.e. "the_answer") as the basics for his or her pondering and brooding. A stupid way to implement the classes consists in having the same code in every philosopher class:

```
class Philosopher1:  
  
    def the_answer(self, *args):  
        return 42  
  
class Philosopher2:  
    def the_answer(self, *args):  
        return 42  
  
class Philosopher3:  
    def the_answer(self, *args):  
        return 42  
  
plato = Philosopher1()  
print(plato.the_answer())  
kant = Philosopher2()  
# let's see what Kant has to say :-)  
print(kant.the_answer())  
  
42  
42
```



We can see that we have multiple copies of the method "the_answer". This is error prone and tedious to maintain, of course.

From what we know so far, the easiest way to accomplish our goal without creating redundant code consists in designing a base, which contains "the_answer" as a method. Each Philosopher class inherits now from this base class:

```

class Answers:
    def the_answer(self, *args):
        return 42

class Philosopher1(Answers):
    pass
class Philosopher2(Answers):
    pass
class Philosopher3(Answers):
    pass
plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())

42
42

```

The way we have designed our classes, each Philosopher class will always have a method "the_answer". Let's assume, we don't know a priori if we want or need this method. Let's assume that the decision, if the classes have to be augmented, can only be made at runtime. This decision might depend on configuration files, user input or some calculations.

```

# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
    return 42

class Philosopher1:
    pass
if required:
    Philosopher1.the_answer = the_answer

class Philosopher2:
    pass
if required:
    Philosopher2.the_answer = the_answer

class Philosopher3:
    pass
if required:
    Philosopher3.the_answer = the_answer

plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-)
if required:

```

```

        print(kant.the_answer())
        print(plato.the_answer())
else:
    print("The silence of the philosophers")

```

Do you need the answer? (y/n): y

42

42

Even though this is another solution to our problem, there are still some serious drawbacks. It's error-prone, because we have to add the same code to every class and it seems likely that we might forget it. Besides this it's getting hardly manageable and maybe even confusing, if there are many methods we want to add.

We can improve our approach by defining a manager function and avoiding redundant code this way. The manager function will be used to augment the classes conditionally.

```

# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
    return 42

# manager function
def augment_answer(cls):
    if required:
        cls.the_answer = the_answer


class Philosopher1:
    pass
augment_answer(Philosopher1)
class Philosopher2:
    pass
augment_answer(Philosopher2)
class Philosopher3:
    pass
augment_answer(Philosopher3)

plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosophers")

```

```
Do you need the answer? (y/n): y
42
42
```

This is again useful to solve our problem, but we, i.e. the class designers, must be careful not to forget to call the manager function "augment_answer". The code should be executed automatically. We need a way to make sure that "some" code might be executed automatically after the end of a class definition.

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
    return 42

def augment_answer(cls):
    if required:
        cls.the_answer = the_answer
    # we have to return the class now:
    return cls

@augment_answer
class Philosopher1:
    pass
@augment_answer
class Philosopher2:
    pass
@augment_answer
class Philosopher3:
    pass

plato = Philosopher1()
kant = Philosopher2()

# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosophers")

Do you need the answer? (y/n): y
42
42
```

Metaclasses can also be used for this purpose as we will learn in the next chapter.

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

METACLASSES

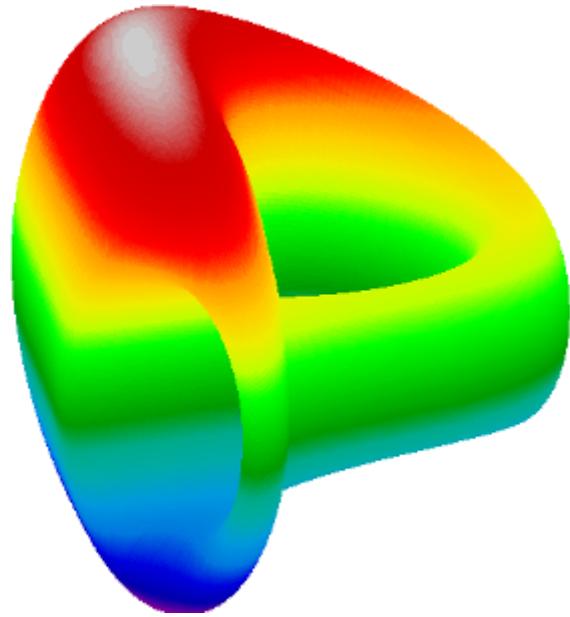
A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.

Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way they implement them. Python is supporting them.

Some programmers see metaclasses in Python as "solutions waiting or looking for a problem".

There are numerous use cases for metaclasses. Just to name a few:

- logging and profiling
- interface checking
- registering classes at creation time
- automatically adding new methods
- automatic property creation
- proxies
- automatic resource locking/synchronization.



DEFINING METACLASSES

Principally, metaclasses are defined like any other Python class, but they are classes that inherit from "type". Another difference is, that a metaclass is called automatically, when the class statement using a metaclass ends. In other words: If no "metaclass" keyword is passed after the base classes (there may be no base classes either) of the class header, type() (i.e. __call__ of type) will be called. If a metaclass keyword is used on the other hand, the class assigned to it will be called instead of type.

Now we create a very simple metaclass. It's good for nothing, except that it will print the content of its arguments in the __new__ method and returns the results of the type.__new__ call:

```
class LittleMeta(type):
    def __new__(cls, classname, superclasses, attributedict):
```

```

        print("classname: ", classname)
        print("superclasses: ", superclasses)
        print("attributedict: ", attributedict)
        return type.__new__(cls, classname, superclasses,
attributedict)
    )

```

We will use the metaclass "LittleMeta" in the following example:

```

class S:
    pass
class A(S, metaclass=LittleMeta):
    pass
a = A()

classname:  A
superclasses:  (<class 'main.S'>,)
attributedict:  {'__module__': '__main__', '__qualname__': 'A'}

```

We can see LittleMeta.__new__ has been called and not type.__new__.

Resuming our thread from the last chapter: We define a metaclass "EssentialAnswers" which is capable of automatically including our augment_answer method:

```

x = input("Do you need the answer? (y/n): ")
if x.lower() == "y":
    required = True
else:
    required = False

def the_answer(self, *args):
    return 42

class EssentialAnswers(type):

    def __init__(cls, classname, superclasses, attributedict):
        if required:
            cls.the_answer = the_answer


class Philosopher1(metaclass=EssentialAnswers):
    pass
class Philosopher2(metaclass=EssentialAnswers):
    pass
class Philosopher3(metaclass=EssentialAnswers):
    pass

plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())

```

```
Do you need the answer? (y/n): y
42
42
```

We have learned in our chapter "Type and Class Relationship" that after the class definition has been processed, Python calls

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

This is not the case, if a metaclass has been declared in the header. That is what we have done in our previous example. Our classes Philosopher1, Philosopher2 and Philosopher3 have been hooked to the metaclass EssentialAnswers. That's why EssentialAnswer will be called instead of type:

```
EssentialAnswer(classname, superclasses, attributes_dict)
```

To be precise, the arguments of the calls will be set to the following values:

```
EssentialAnswer('Philosopher1',
                 (),
                 {'__module__': '__main__', '__qualname__':
                  'Philosopher1'})
```

The other philosopher classes are treated in an analogue way.

CREATING SINGLETONS USING METACLASSES

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. It is used in cases where exactly one object is needed. The concept can be generalized to restrict the instantiation to a certain or fixed number of objects. The term stems from mathematics, where a singleton, - also called a unit set -, is used for sets with exactly one element.

```
class Singleton(type):
    instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls.instances:
            cls.instances[cls] = super(Singleton,
cls).__call__(*args, **kwargs)
        return cls.instances[cls]

class SingletonClass(metaclass=Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
```

```
x = RegularClass()
y = RegularClass()
print(x == y)
```

True
False

CREATING SINGLETONS USING METACLASSES

Alternatively, we can create Singleton classes by inheriting from a Singleton class, which can be defined like this:

```
class Singleton(object):
    instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class SingletonClass(Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
x = RegularClass()
y = RegularClass()
print(x == y)

True
False
```

COUNT METHOD CALLS USING A METACLASS

INTRODUCTION

After you have hopefully gone through our chapter [Introduction into Metaclasses](#) you may have asked yourself about possible use cases for metaclasses. There are some interesting use cases and it's not - like some say - a solution waiting for a problem. We have mentioned already some examples.

In this chapter of our tutorial on Python, we want to elaborate an example metaclass, which will decorate the methods of the subclass. The decorated function returned by the decorator makes it possible to count the number of times each method of the subclass has been called.

This is usually one of the tasks, we expect from a profiler. So we can use this metaclass for simple profiling purposes. Of course, it will be easy to extend our metaclass for further profiling tasks.



PRELIMINARY REMARKS

Before we actually dive into the problem, we want to call to mind again how we can access the attributes of a class. We will demonstrate this with the list class. We can get the list of all the non private attributes of a class - in our example the random class - with the following construct.

```
import random
cls = "random" # name of the class as a string
all_attributes = [x for x in dir(eval(cls)) if not
x.startswith("__") ]
print(all_attributes)

['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType',
'_MethodType', '_Sequence', '_Set', '_acos', '_ceil', '_cos', '_e',
```

```
'__exp__', '__inst__', '__log__', '__pi__', '__random__', '__sha512__', '__sin__',
'__sqrt__', '__test__', '__test_generator__', '__urandom__', '__warn__',
'__betavariate__', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed',
'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

Now, we are filtering the callable attributes, i.e. the public methods of the class.

```
methods = [x for x in dir(eval(cls)) if not x.startswith("__")
           and callable(eval(cls + "." + x))]
print(methods)

['Random', 'SystemRandom', '__BuiltinMethodType', '__MethodType',
 '__Sequence', '__Set', '__acos__', '__ceil__', '__cos__', '__exp__', '__log__',
 '__sha512__', '__sin__', '__sqrt__', '__test__', '__test_generator__', '__urandom__',
 '__warn__', '__betavariate__', 'choice', 'expovariate', 'gammavariate',
 'gauss', 'getrandbits', 'getstate', 'lognormvariate',
 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange',
 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
 'vonmisesvariate', 'weibullvariate']
```

Getting the non callable attributes of the class can be easily achieved by negating callable, i.e. adding "not":

```
non_callable_attributes = [x for x in dir(eval(cls)) if not
                           x.startswith("__")
                           and not callable(eval(cls + "." + x))]
print(non_callable_attributes)

['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'SG_MAGICCONST',
 'TWOPI', '__e', '__inst__', '__pi__', '__random__']
```

In normal Python programming it is neither recommended nor necessary to apply methods in the following way, but it is possible:

```
lst = [3,4]
list.__dict__["append"](lst, 42)
lst
```

The previous Python code returned the following output:

```
[3, 4, 42]
```

Please note the remark from the Python documentation: "Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are

not in the result list when the argument is a class."

A DECORATOR FOR COUNTING FUNCTION CALLS

Finally, we will begin to design the metaclass, which we have mentioned as our target in the beginning of this chapter. It will decorate all the methods of its subclass with a decorator, which counts the number of calls. We have defined such a decorator in our chapter [Memoization and Decorators](#):

```
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0
    helper.__name__ = func.__name__
    return helper
```

We can use it in the usual way:

```
@call_counter
def f():
    pass
print(f.calls)
for _ in range(10):
    f()

print(f.calls)

0
10
```

It better if you call to mind the alternative notation for decorating function. We will need this in our final metaclass:

```
def f():
    pass
f = call_counter(f)
print(f.calls)
for _ in range(10):
    f()

print(f.calls)

0
10
```

THE "COUNT CALLS" METACLASS

Now we have all the necessary "ingredients" together to write our metaclass. We will include our `call_counter` decorator as a staticmethod:

```
class FuncCallCounter(type):
    """ A Metaclass which decorates all the methods of the
        subclass using call_counter as the decorator
    """

    @staticmethod
    def call_counter(func):
        """ Decorator for counting the number of function
            or method calls to the function or method func
        """
        def helper(*args, **kwargs):
            helper.calls += 1
            return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__ = func.__name__

        return helper

    def __new__(cls, classname, superclasses, attributedict):
        """ Every method gets decorated with the decorator
        call_counter,
            which will do the actual call counting
        """
        for attr in attributedict:
            if callable(attributedict[attr]) and not
attr.startswith("__"):
                attributedict[attr] =
cls.call_counter(attributedict[attr])

        return type.__new__(cls, classname, superclasses,
attributedict)

class A(metaclass=FuncCallCounter):

    def foo(self):
        pass

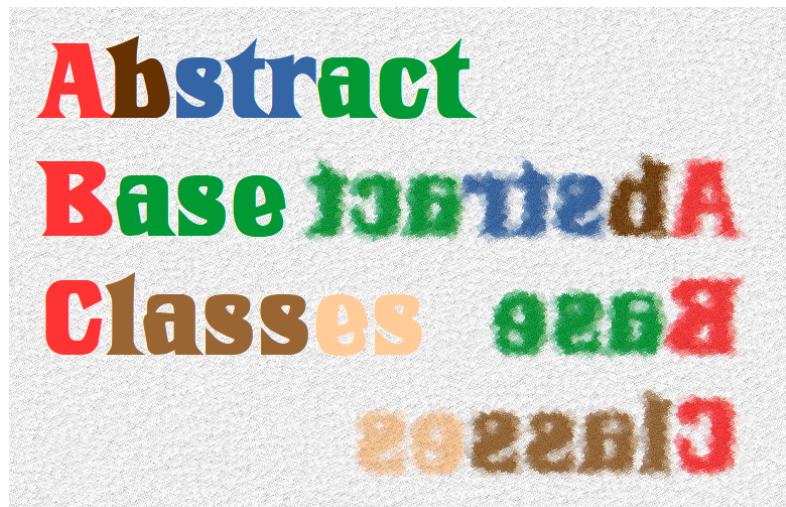
    def bar(self):
        pass
if __name__ == "__main__":
    x = A()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    x.bar()
    print(x.foo.calls, x.bar.calls)
```

```
0 0  
1 0  
2 1
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu
by Bernd Klein

ABSTRACT CLASSES

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.



Let's look at the following example:

```
class AbstractClass:

    def do_something(self):
        pass

class B(AbstractClass):
    pass
a = AbstractClass()
b = B()
```

If we start this program, we see that this is not an abstract class, because:

- we can instantiate an instance from
- we are not required to implement do_something in the class definition of B

Our example implemented a case of simple inheritance which has nothing to do with an abstract class. In fact, Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - for obvious reasons - abc.

The following Python code uses the abc module and defines an abstract base class:

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
```

```

    self.value = value
    super().__init__()

    @abstractmethod
    def do_something(self):
        pass

```

We will define now a subclass using the previously defined abstract class. You will notice that we haven't implemented the `do_something` method, even though we are required to implement it, because this method is decorated as an abstract method with the decorator "`abstractmethod`". We get an exception that `DoAdd42` can't be instantiated:

```

class DoAdd42(AbstractClassExample):
    pass
x = DoAdd42(4)

```

The previous Python code returned the following output:

```

-----
-----
TypeError                                 Traceback (most recent
call last)
<ipython-input-9-83fb8cead43d> in <module>()
      2     pass
      3
----> 4 x = DoAdd42(4)
TypeError: Can't instantiate abstract class DoAdd42 with abstract
methods do_something

```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```

class DoAdd42(AbstractClassExample):
    def do_something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):

    def do_something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)
print(x.do_something())
print(y.do_something())

52
420

```

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

You may think that abstract methods can't be implemented in the abstract base class. This impression is wrong: An abstract method can have an implementation in the abstract class! Even if they are implemented, designers of subclasses will be forced to override the implementation. Like in other cases of "normal" inheritance, the abstract method can be invoked with super() call mechanism. This makes it possible to provide some basic functionality in the abstract method, which can be enriched by the subclass implementation.

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
    def do_something(self):
        super().do_something()
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

```
Some implementation!
The enrichment from AnotherSubclass
```

© 2011 - 2018, Bernd Klein, Bodenseo; Design by Denise Mitchinson adapted for python-course.eu by Bernd Klein