Introduction to Python Language

1. Overview

This is an introduction course to Python programming language for students with a little bit of programming experience who want to learn Python.

The references of this course are the following (a short list, you can find lot of references):

- https://developers.google.com/edu/python
- https://cs.stanford.edu/people/nick/py/
- http://www.spronck.net/pythonbook/pythonbook.pdf
- https://www.python-course.eu/index.php

The course is organized as follows:

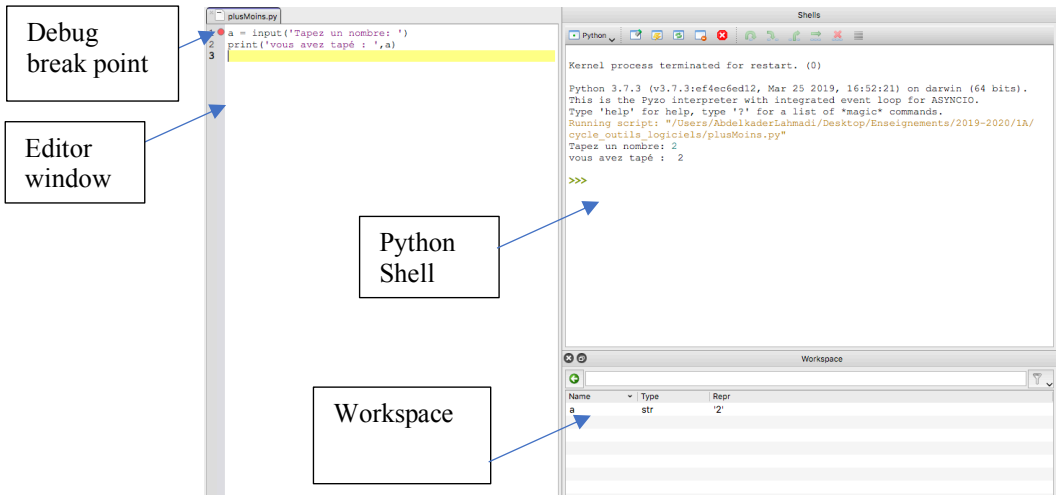
- Class 1 (2h): Python Set up and Intro, Expressions and variables
- Class 2 (2h): Conditions, iterations (while and for), functions
- Class 3 (2h): Strings, lists, dictionaries
- Class 4 (2h): read and write files, pandas, CSV files
- Class 5 (2h): numerical programming (numpy and matplotlib)

2. Setup and Introduction

Python has become a language of choice for teaching people how to program. It is a powerful language, that is easy to use, and that offers all the possibilities that any other computer language offers. It is easily portable between different operating systems. It is freely available. For beginning programmers, it has the advantage that it enforces writing readable code. Python is also a language that is used for many practical applications, either as a basis for complete programs, or as an extension to programs written in a different language.

To run Python programs, you need a "Python interpreter." Fortunately, Python interpreters are freely available for almost every machine in existence. Visit http://www.python.org to download a Python interpreter for your computer. Make sure that you get a Python 3 interpreter. Install it. After the installation finishes, in principle you are ready to write and run Python programs.

A Python program is just a text file that you edit directly. You can find many editors for Python code, in particular in this course we will Pyzo available at: https://pyzo.org/start.html
After installing Pyzo, you ca run it and you get a graphical interface to edit and run you Python code.



Introduction to Python Language

Python instructions

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.
The set of Python built-ins is the following:

```
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
```

Indentation

One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning. A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or whatever. If one of the lines in a group has a different indentation, it is flagged as a syntax error. Python's use of whitespace feels a little strange at first, but it's logical and I found I got used to it very quickly. Avoid using TABs as they greatly complicate the indentation scheme (not to mention TABs may mean different things on different platforms). Set your editor to insert spaces instead of TABs for Python code.

The pyzo code editor is able to auto-indent and organize you code, i.e., if, for instance, you write the first line of an if statement, once you press Enter to go to the next line, it will automatically “jump in” one level of indentation (if it does not, it is very likely that you forgot the colon at the end of the conditional expression). Also, when you have indented one line to a certain level of indentation, the next line will use the same level. You can get rid of indentations using the Backspace key

Variable Names

Since Python variables don't have any type spelled out in the source code, it's extra helpful to give meaningful names to your variables to remind yourself of what's going on. So use "name" if it's a single name, and "names" if it's a list of names, and "tuples" if it's a list of tuples. Many basic Python errors result from forgetting what type of value is in each variable, so use your variable names (all you have really) to help keep things straight.
As far as actual naming goes, some languages prefer underscored_parts for variable names made up of "more than one word," but other languages prefer camelCasing. In general, Python prefers the underscore method but guides developers to defer to camelCasing if integrating into existing Python code that already uses that style. Readability counts. Read more in the section on naming conventions in PEP 8.
As you can guess, keywords like 'print' and 'while' cannot be used as variable names — you'll get a syntax error if you do. However, be careful not to use built-ins as variable names. For example, while 'str' and 'list' may seem like good names, you'd be overriding those system variables. Built-ins are not keywords and thus, are susceptible to inadvertent use by new Python developers.

Class 1
Starting with Python

THE INTERPRETER, AN INTERACTIVE SHELL

THE TERMS 'INTERACTIVE' AND 'SHELL'

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

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

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

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

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

USING THE PYTHON INTERACTIVE SHELL

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

```
python
```

Python comes back with the following information:

```
$ python
Python 2.7.11 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more informatio
n.
```

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

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

```
bernd@venus:~$ $ python
python             python3            python3.7-config   python-confi
g
python2            python3.6          python3.7m         pythontex
python2.7          python3.6-config   python3.7m-config  pythontex3
python2.7-config   python3.6m         python3-config     python-white
board
python2-config     python3.6m-config  python3m
python2-pbr        python3.7          python3m-config
bernd@venus:~$ python
```

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

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

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

```
hello
```

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

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

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

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

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

```
Hello World
```

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

```
"Hello World"
```

Output:

```
'Hello World'
```

```
3
```

Output:

```
3
```

HOW TO QUIT THE PYTHON SHELL

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

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

THE SHELL AS A SIMPLE CALCULATOR

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

```
4.567 * 8.323 * 17
```

Output:

```
646.189397
```

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

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

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

```
3 + 2 * 4
```

Output:

```
11
```

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

```
_
```

Output:

```
11
```

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

```
_ * 3
```

Output:

```
33
```

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

USING VARIABLES

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

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

```
30
```

MULTILINE STATEMENTS

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

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

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

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

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

STRINGS

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

```
"Hello" + " " + "World"
```

Output:

```
'Hello World'
```

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

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

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

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

```
".-." * 4
```

Output:

```
'.-..-..-..-.'
```

STRUCTURING WITH INDENTATION

BLOCKS

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

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

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

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

- begin ... end

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

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

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

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

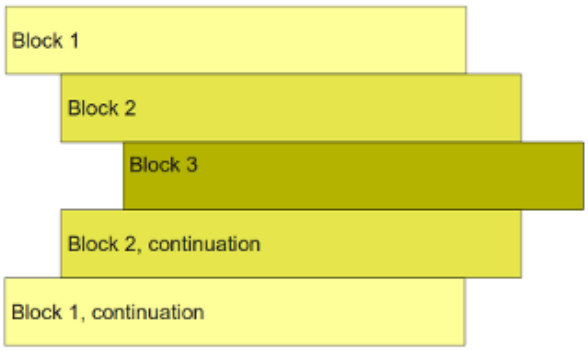
The indentations in this code fragment are not necessary. So the code could be written - offending common decency - as

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

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

INDENTING CODE

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



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

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

```
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. All in all, Python structures by colons and indentation.

DATA TYPES AND VARIABLES

INTRODUCTION

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

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

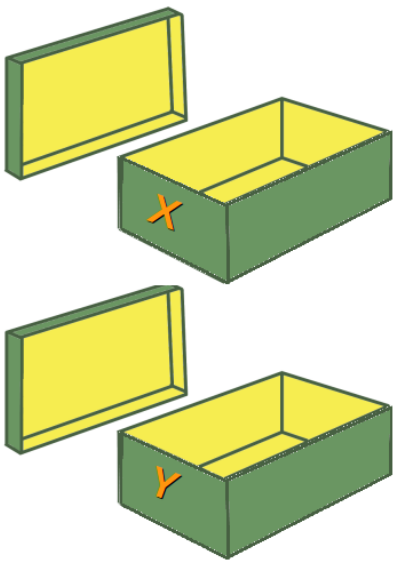
So, this chapter is worth reading both for beginners and for advanced programmers in other programming languages.

VARIABLES

GENERAL CONCEPT OF VARIABLES

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

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



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

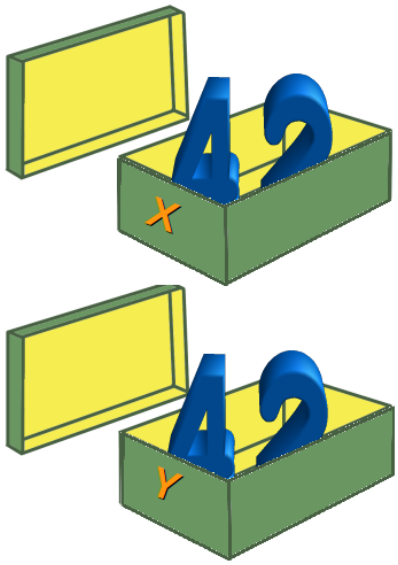
What we have said so far about variables best fits the way variables are implemented in C, C++ or Java. Variable names have to be declared in these languages before they can be used.

```
int x;

int y;
```

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

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

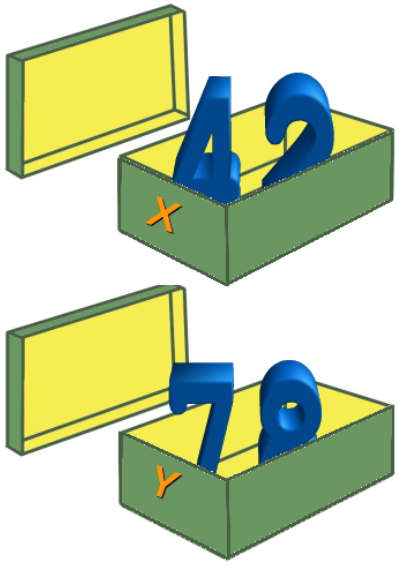
We will assign the value 42 to both variables and we can see that two numbers are physically saved in the memory, which correspond to the two shoeboxes in the following picture.




```
x = 42;

y = 42;
```

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

```
y = 78;
```



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

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

VARIABLES IN PYTHON

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

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

```
i = 42
```

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

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

```
43
```

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

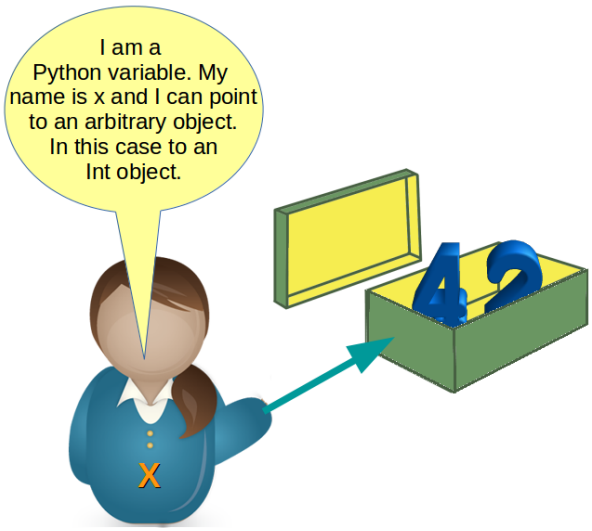
```
i = 42          # data type is implicitly set to integer
i = 42 + 0.11       # data type is changed to float
i = "forty"     # and now it will be a string
```

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

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

OBJECT REFERENCES

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



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

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

```
x = 42
print(x)
```

```
42
```

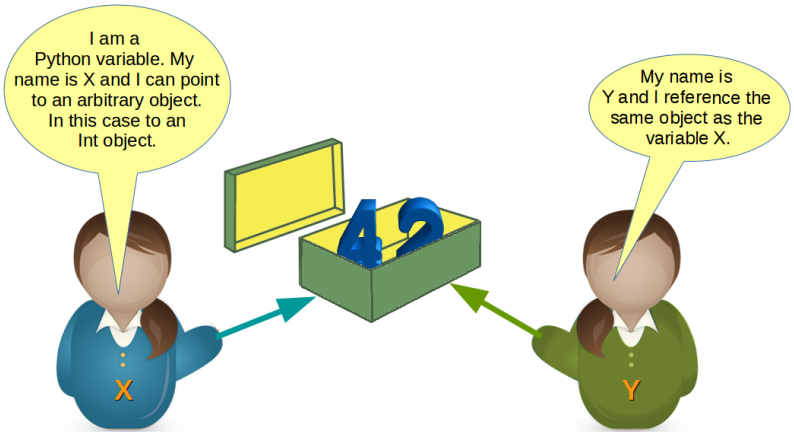
```
x = "Now x references a string"
print(x)
```

```
Now x references a string
```

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

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

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

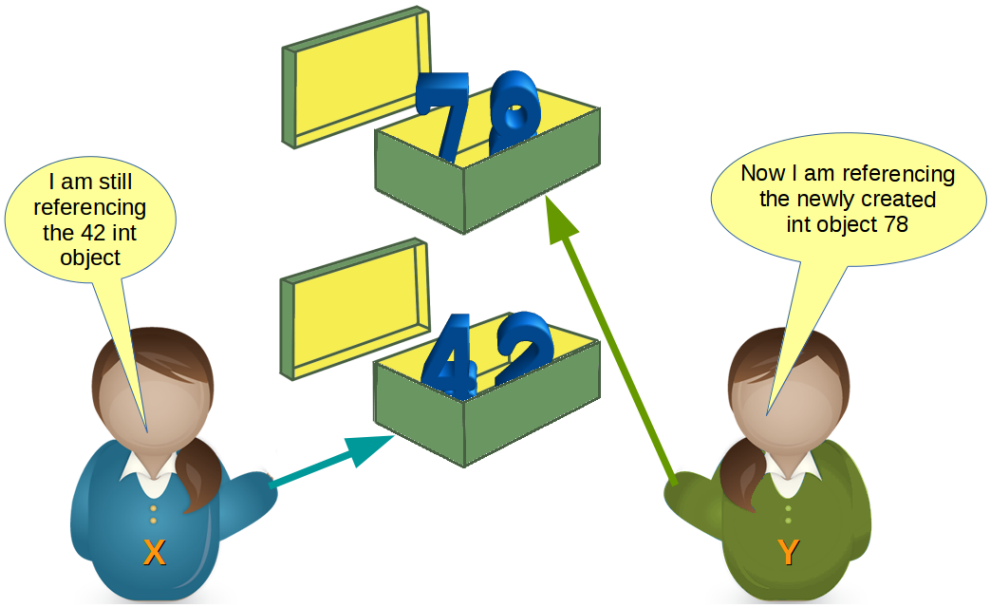


What will happen when we execute

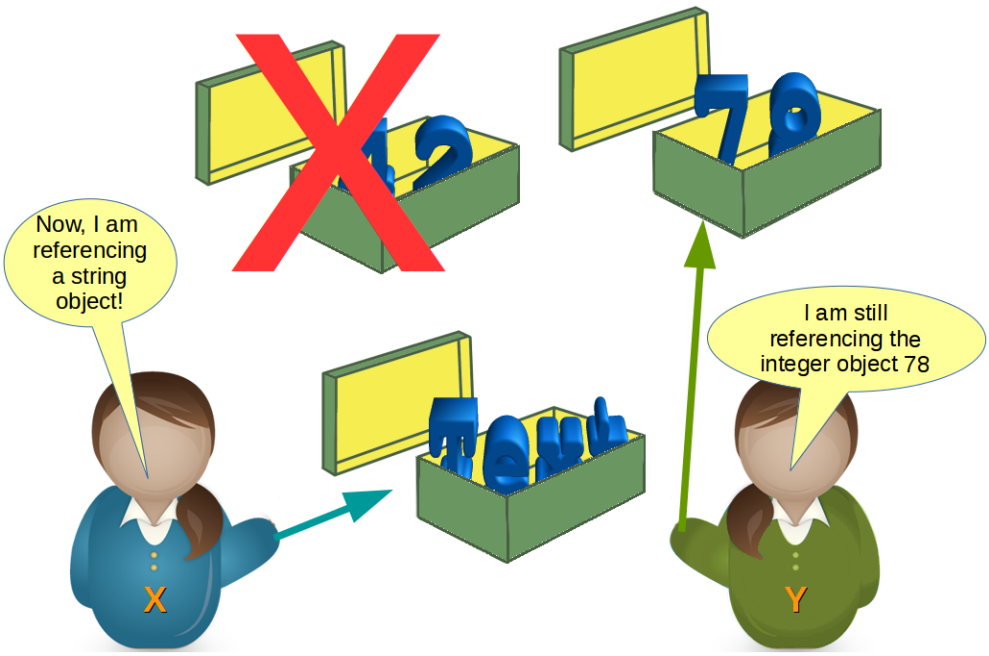
```
y = 78
```

after the previous code?

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



Most probably, we will see further changes to the variables in the flow of our program. There might be, for example, a string assignment to the variable x. The previously integer object "42" will be orphaned after this assignment. It will be removed by Python, because no other variable is referencing it.



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

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

```
x = 42
id(x)
```

Output:

```
140709828470448
```

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

Output:

```
(140709828470448, 140709828470448)
```

```
y = 78
id(x), id(y)
```

Output:

```
(140709828470448, 140709828471600)
```

VALID VARIABLE NAMES

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

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

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

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

The following variable definitions are all valid:

```
height = 10
maximum_height = 100

υψος = 10
μεγιστη_υψος = 100

MinimumHeight = 1
```

PYTHON KEYWORDS

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

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

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

```
help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely 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 writin
g
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, typ
e
"modules", "keywords", "symbols", or "topics".  Each module also come
s
with a one-line summary of what it does; to list the modules whose na
me
or summary contain a given string such as "spam", type "modules spa
m".

help>
```

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

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get 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 'I' (like in "Indiana") as single character variable names. They should be avoided, because these characters are indistinguishable from the numerals one and zero in some fonts. When tempted to use 'l', use 'L' instead, if you cannot think of a better name anyway. The Style Guide has to say the following about the naming of identifiers in standard modules: "All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names."

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

CHANGING DATA TYPES AND STORAGE LOCATIONS

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

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

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

NUMBERS

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

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

```
a = 0o10

print(a)
```

```
8
```

·Hexadecimal literals (base 16)
Hexadecimal literals have to be prefixed either by "0x" or "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 pr
efixed by a leading "0", followed by a "b" or "B":

```
x = 0b101010
x
```

Output:

```
42
```

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

```
x = hex(19)
x
```

Output:

```
'0x13'
```

```
type(x)
```

Output:

```
str
```

```
x = bin(65)
x
```

Output:

```
'0b1000001'
```

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

Output:

```
'0o101'
```

```
oct(0b101101)
```

Output:

```
'0o55'
```

Integers in Python3 can be of unlimited size

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

```
787366098712738903245678234782358292837498729182728
```

```
x * x * x
```

Output:

```
488123970070638215986770162105731315538827586091948617997871122950228891123960901918308618286311523282239313708275589787123005317148968569797875581092352
```

- **Long integers**

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

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

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

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

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

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

- **Floating-point numbers**

For example: 42.11, 3.1415e-10

- **Complex numbers**

Complex numbers are written as

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

Examples:

```
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-explanatory:

```
10 / 3
```

Output:

```
3.3333333333333335
```

```
10.0 / 3.0
```

Output:

```
3.3333333333333335
```

```
10.5 / 3.5
```

Output:

```
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 returns the greatest integer floor (x) = ⌊ x ⌋ that is less than or equal to x.

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

```
9 // 3
```

Output:

```
3
```

```
10 // 3
```

Output:

```
3
```

```
11 // 3
```

Output:

```
3
```

```
12 // 3
```

Output:

```
4
```

```
10.0 // 3
```

Output:

```
3.0
```

```
-7 // 3
```

Output:

```
-3
```

```
-7.0 // 3
```

Output:

```
-3.0
```

STRINGS

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

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

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

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

You can transform it like this in Python:

```
hex(65)
```

Output:

```
'0x41'
```

```
int(0x41)
```

Output:

```
65
```

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

UNICODE ENCODINGS

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

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:

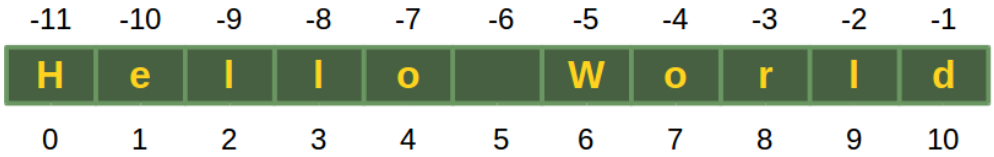
```
txt = "He said: \"It doesn't matter, if you enclose a string in sin
gle or double quotes!\""
print(txt)
```

```
He said: "It doesn't matter, if you enclose a string in sin
gle 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 ".)



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

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

Output:

```
'H'
```

```
s[5]
```

Output:

```
' '
```

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

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

Output:

```
'd'
```

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

```
s[-1]
```

Output:

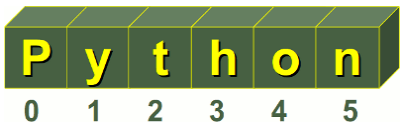
```
'd'
```

```
s[-2]
```

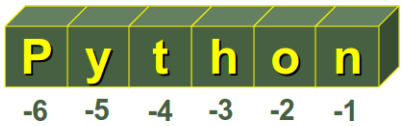
Output:

```
'l'
```

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



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



SOME OPERATORS AND FUNCTIONS FOR STRINGS

- `Concatenation`

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

- `Repetition`

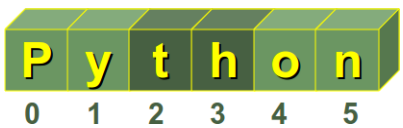
String can be repeated or repeatedly concatenated with the asterisk operator "″: ″-″ 3 will result in "---"

- `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] = "."
```

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

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

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

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

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

A STRING PECULIARITY

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

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

```
a = "Linux"
b = "Linux"
a is b
```

Output:

```
True
```

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

```
a = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
b = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
a is b
```

Output:

```
True
```

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

```
a = "Baden-Württemberg"
b = "Baden-Württemberg"
a is b
```

Output:

```
False
```

```
a == b
```

Output:

```
True
```

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

```
a = "Baden!"
b = "Baden!"
a is b
```

Output:

```
False
```

```
a = "Baden1"
b = "Baden1"
a is b
```

Output:

```
True
```

ESCAPE SEQUENCES IN STRINGS

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

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

BYTE STRINGS

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

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

```
b'Hallo'
```

ARITHMETIC AND COMPARISON OPERATORS

INTRODUCTION

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

OPERATORS

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

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

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

//	Truncation Division (also known as floordivision or floor division)
	The result of this division is the integral part of the result, i.e. the fractional part is truncated, if there is any.
	It works both for integers and floating-point numbers, but there is a difference between the type of the results: If both the dividend and the divisor are integers, the result will also be an integer. If either the divident or the divisor is a float, the result will be the truncated result as a float.
		`10 // 3`
		`3`
		If at least one of the operands is a float value, we get a truncated float value as the result.
		`10.0 // 3`
		`3.0`
		`>>>`
		A note about efficiency:
		The results of int(10 / 3) and 10 // 3 are equal. But the "//" division is more than two times as fast! You can see this here:
		`In [9]: %%timeit`
		`for x in range(1, 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`

+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

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 + "!")
```

```
What's your name? Melisa
Nice to meet you Melisa!
```

```
age = input("Your age? ")
print("So, you are already " + age + " years old, " + name + "!")
```

```
Your age? 26
So, you are already 26 years old, Melisa!
```

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

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

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

```
cities_canada = input("Largest cities in Canada: ")
print(cities_canada, type(cities_canada))
```

```
Largest cities in Canada: Toronto, Calgary, Montreal, Ottaw
a
Toronto, Calgary, Montreal, Ottawa <class 'str'>
```

```
cities_canada = eval(input("Largest cities in Canada: "))
print(cities_canada, type(cities_canada))
```

```
Largest cities in Canada: Toronto, Calgary, Montreal, Ottaw
a
```

```
---------------------------------------------------------------
---------------
NameError                                 Traceback (most r
ecent call last)
<ipython-input-3-4febe85dfcfd> in <module>
----> 1 cities_canada = eval(input("Largest cities in Canad
a: "))
      2 print(cities_canada, type(cities_canada))

<string> in <module>

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

```
population = input("Population of Toronto? ")
print(population, type(population))
```

```
Population of Toronto? 2615069
2615069 <class 'str'>
```

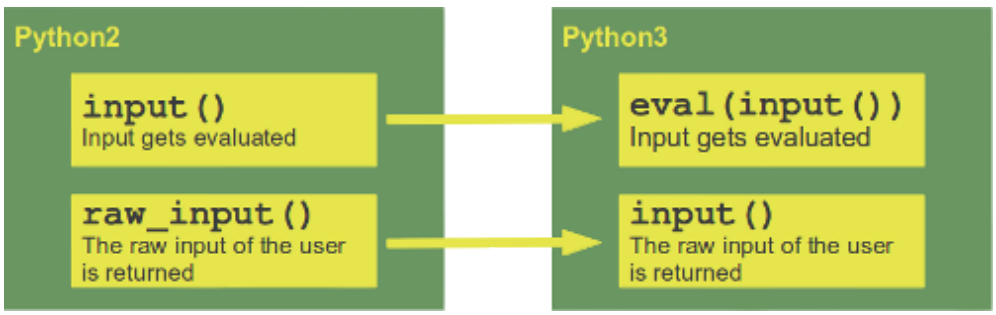
```
population = int(input("Population of Toronto? "))
print(population, type(population))
```

```
Population of Toronto? 2615069
2615069 <class 'int'>
```

DIFFERENCES BETWEEN PYTHON2 AND PYTHON3

The usage of input or better the implicit evaluation of the input has often lead to serious programming mistakes in the earlier Python versions, i.e. 2.x Therefore, in Python 3 the input function behaves like the raw_input function from Python2.

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



PRINT

INTRODUCTION

In principle, every computer program has to communicate with the environment or the "outside world". For this purpose nearly every programming language has special I/O functionalities, i.e. input/output. This ensures the interaction or communication with the other components e.g. a database or a user. Input often comes - as we have already seen - from the keyboard and the corresponding Python command, or better, the corresponding Python function for reading from the standard input is input().

We have also seen in previous examples of our tutorial that we can write into the standard output by using print. In this chapter of our tutorial we want to have a detailed look at the print function. As some might have skipped over it, we want to emphasize that we wrote "print function" and not "print statement". You can easily find out how crucial this difference is, if you take an arbitrary Python program written in version 2.x and if you try to let it run with a Python3 interpreter. In most cases you will receive error messages. One of the most frequently occurring errors will be related to print, because most programs contain prints. We can generate the most typical error in the interactive Python shell:

```
print 42
  File "", line 1
    print 42

  File "<ipython-input-1-5d54469f7ddf>", line 1
    print 42
          ^
SyntaxError: Missing parentheses in call to 'print'. Did yo
u mean print(42)?
```

This is a familiar error message for most of us: We have forgotten the parentheses. "print" is - as we have already mentioned - a function in version 3.x. Like any other function print expects its arguments to be surrounded by parentheses. So parentheses are an easy remedy for this error:

```
print(42)
```

```
42
```

But this is not the only difference to the old print. The output behaviour has changed as well.

PRINT FUNCTION

The arguments of the print function are the following ones:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The print function can print an arbitrary number of values ("value1, value2, ..."), which are separated by commas. These values are separated by a space in the output. In the following example, we can see two print calls. We are printing two values in both cases, i.e. a string and a float number:

```
a =  3.564
print("a = ", a)
```

```
a =  3.564
```

```
print("a = \n", a)
```

```
a =
 3.564
```

We can learn from the second print of the example that a blank between two values, i.e. "a = \textbackslash n" and "3.564", is always printed, even if the output is continued in the following line. This is different from Python 2, as there will be no blank printed, if a new line has been started. It's possible to redefine the seperator between values by assigning an arbitrary string to the keyword parameter "sep", i.e. an empty string or a smiley:

```
print("a","b")
```

```
a b
```

```
print("a","b",sep="")
```

```
ab
```

```
print(192,168,178,42,sep=".")
```

```
192.168.178.42
```

```
print("a","b",sep=":-)")
```

```
a:-)b
```

A print call is ended by a newline, as we can see in the following usage:

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

To change this behaviour, we can assign an arbitrary string to the keyword parameter "end". This string will be used for ending the output of the values of a print call:

```
for i in range(4):
    print(i, end=" ")
```

```
0 1 2 3
```

```
for i in range(4):
     print(i, end=" :-) ")
```

```
0 :-) 1 :-) 2 :-) 3 :-)
```

The output of the print function is send to the standard output stream (sys.stdout) by default. By redefining the keyword parameter "file" we can send the output into a different stream e.g. sys.stderr or a file:

```
fh = open("data.txt","w")
print("42 is the answer, but what is the question?", file=fh)
fh.close()
```

We can see that we don't get any output in the interactive shell. The output is sent to the file "data.txt". It's also possible to redirect the output to the standard error channel this way:

```
import sys
# output into sys.stderr:

print("Error: 42", file=sys.stderr)
```

```
Error: 42
```

FORMATTED OUTPUT

MANY WAYS FOR A NICER OUTPUT

In this chapter of our Python tutorial we will have a closer look at the various ways of creating nicer output in Python. We present all the different ways, but we recommend that you should use the format method of the string class, which you will find at end of the chapter. "string format" is by far the most flexible and Pythonic approach.

So far we have used the print function in two ways, when we had to print out more than two values:

The easiest way, but not the most elegant one:

We used print with a comma separated list of values to print out the results, as we can see in the following example. All the values are separated by blanks, which is the default behaviour. We can change the default value to an arbitrary string, if we assign this string to the keyword parameter "sep" of the print function:

```
q = 459
p = 0.098
print(q, p, p * q)
```

```
459 0.098 44.982
```

```
print(q, p, p * q, sep=",")
```

```
459,0.098,44.982
```

```
print(q, p, p * q, sep=" :-) ")
```

```
459 :-) 0.098 :-) 44.982
```

Alternatively, we can construe a new string out of the values by using the string concatenation operator:

```
print(str(q) + " " + str(p) + " " + str(p * q))
```

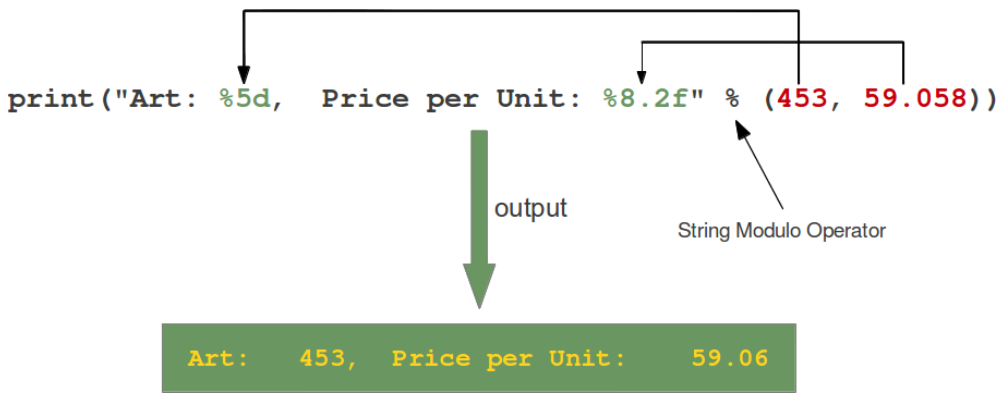
```
459 0.098 44.982
```

The second method is inferior to the first one in this example.

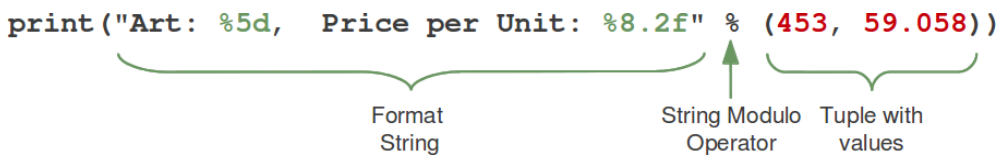
THE OLD WAY OR THE NON-EXISTING PRINTF AND SPRINTF

Is there a printf in Python? A burning question for Python newbies coming from C, Perl, Bash or other programming languages who have this statement or function. To answer "Python has a print function and no printf function" is only one side of the coin or half of the truth. One can go as far as to say that this answer is not true. So there is a "printf" in Python? No, but the functionality of the "ancient" printf is contained in Python. To this purpose the modulo operator "%" is overloaded by the string class to perform string formatting. Therefore, it is often called string modulo (or somethimes even called modulus) operator, though it has not a lot in common with the actual modulo calculation on numbers. Another term for it is "string interpolation", because it interpolates various class types (like int, float and so on) into a formatted string. In many cases the string created via the string interpolation mechanism is used for outputting values in a special way. But it can also be used, for example, to create the right format to put the data into a database. Since Python 2.6 has been introduced, the string method format should be used instead of this old-style formatting. Unfortunately, string modulo "%" is still available in Python3 and what is even worse, it is still widely used. That's why we cover it in great detail in this tutorial. You should be capable of understanding it, when you encounter it in some Python code. However, it is very likely that one day this old style of formatting will be removed from the language. So you should get used to str.format().

The following diagram depicts how the string modulo operator works:



On the left side of the "string modulo operator" there is the so-called format string and on the right side is a tuple with the content, which is interpolated in the format string. The values can be literals, variables or arbitrary arithmetic expressions.



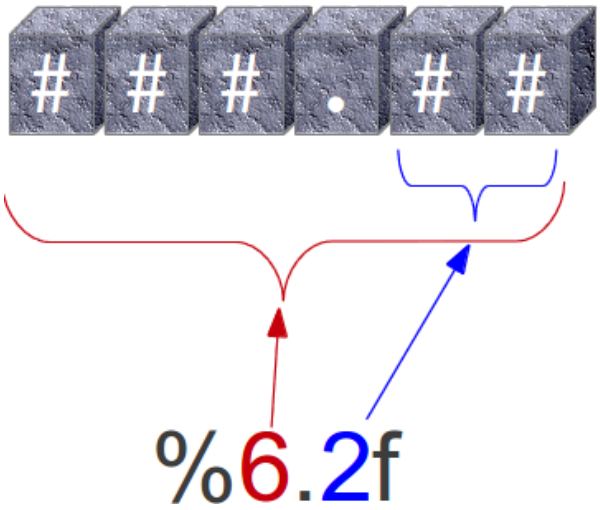
The format string contains placeholders. There are two of those in our example: "%5d" and "%8.2f".

The general syntax for a format placeholder is

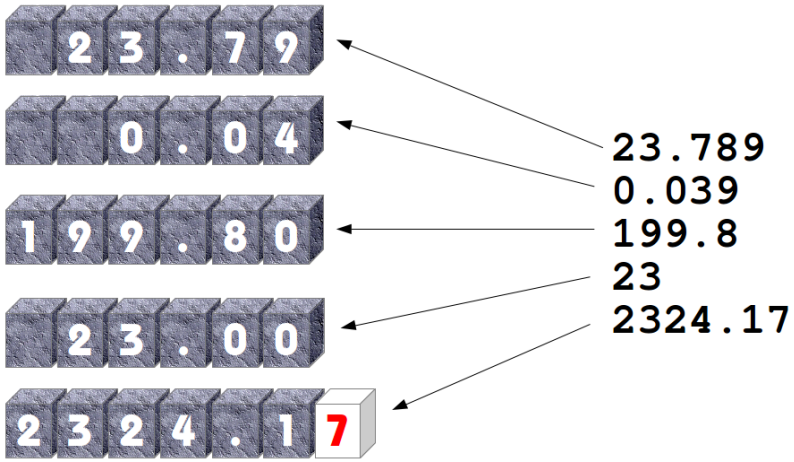
```
%[flags][width][.precision]type
```

Let's have a look at the placeholders in our example. The second one "%8.2f" is a format description for a float number. Like other placeholders, it is introduced with the "%" character. This is followed by the total number of digits the string should contain. This number includes the decimal point and all the digits, i.e. before and after the decimal point. Our float number 59.058 has to be formatted with 8 characters. The decimal part of the number or the precision is set to 2, i.e. the number following the "." in our placeholder. Finally, the last character "f" of our placeholder stands for "float".

If you look at the output, you will notice that the 3 decimal digits have been rounded. Furthermore, the number has been preceded in the output with 3 leading blanks.



The first placeholder "%5d" is used for the first component of our tuple, i.e. the integer 453. The number will be printed with 5 characters. As 453 consists only of 3 digits, the output is padded with 2 leading blanks. You may have expected a "%5i" instead of "%5d". Where is the difference? It's easy: There is no difference between "d" and "i" both are used for formatting integers. The advantage or beauty of a formatted output can only be seen, if more than one line is printed with the same pattern. In the following picture you can see, how it looks, if five float numbers are printed with the placeholder "%6.2f" in subsequent lines:



Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Obsolete and equivalent to 'd', i.e. signed integer decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using repr()).
s	String (converts any python object using str()).
%	No argument is converted, results in a "%" character in the result.

The following examples show some example cases of the conversion rules from the table above:

```
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: %
```

Examples:

```
 0X2F

   2F

0X002F

 0o31

+42

 42

+42

 42

42
```

Even though it may look so, the formatting is not part of the print function. If you have a closer look at our examples, you will see that we passed a formatted string to the print function. Or to put it in other words: If the string modulo operator is applied to a string, it returns a string. This string in turn is passed in our examples to the print function. So, we could have used the string modulo functionality of Python in a two layer approach as well, i.e. first create a formatted string, which will be assigned to a variable and this variable is passed to the print function:

```
Price: $   356.09
```

THE PYTHONIC WAY: THE STRING METHOD "FORMAT"

The Python help function is not very helpful concerning the string format method. All it says is this:

```
 |  format(...)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from a
rgs and kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
```

Let's dive into this topic a little bit deeper: The format method was added in Python 2.6. The general form of this method looks like this:

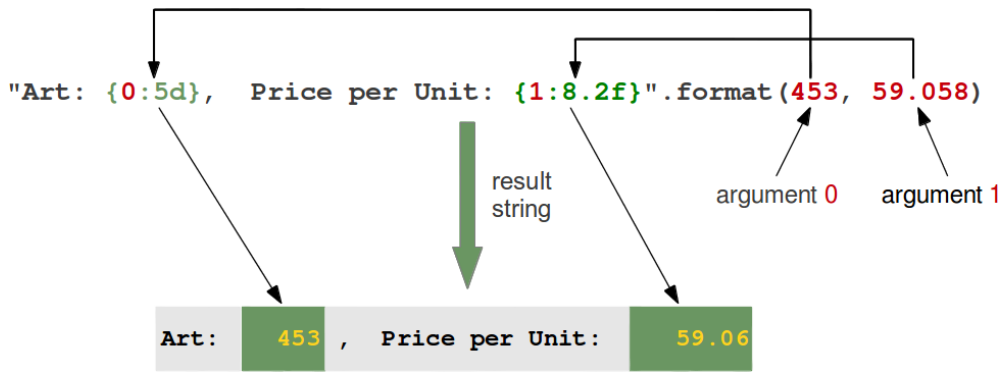
```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

The template (or format string) is a string which contains one or more format codes (fields to be replaced) embedded in constant text. The "fields to be replaced" are surrounded by curly braces {}. The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments, according to the rules which we will specify soon. Anything else, which is not contained in curly braces will be literally printed, i.e. without any changes. If a brace character has to be printed, it has to be escaped by doubling it: {{ and }}.

There are two kinds of arguments for the .format() method. The list of arguments starts with zero or more positional arguments (p0, p1, ...), it may be followed by zero or more keyword arguments of the form name=value.

A positional parameter of the format method can be accessed by placing the index of the parameter after the opening brace, e.g. {0} accesses the first parameter, {1} the second one and so on. The index inside of the curly braces can be followed by a colon and a format string, which is similar to the notation of the string modulo, which we had discussed in the beginning of the chapter of our tutorial, e.g. {0:5d} If the positional parameters are used in the order in which they are written, the positional argument specifiers inside of the braces can be omitted, so '{} {} {}' corresponds to '{0} {1} {2}'. But they are needed, if you want to access them in different orders: '{2} {1} {0}'.

The following diagram with an example usage depicts how the string method "format" works works for positional parameters:



Examples of positional parameters:

Output:

```
'First argument: 47, second one: 11'
```

Output:

```
'Second argument: 11, first one: 47'
```

Output:

```
'Second argument:  11, first one:   47.42'
```

Output:

```
'First argument: 47, second one: 11'
```

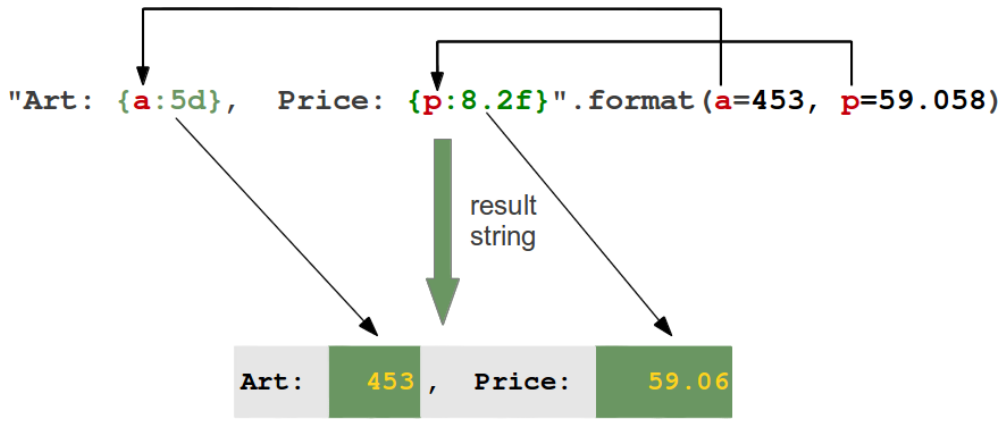
Output:

```
'various precisions:   1.41 or  1.415'
```

In the following example we demonstrate how keyword parameters can be used with the format method:

Output:

```
'Art:   453,  Price:    59.06'
```



It's possible to "left or right justify" data with the format method. To this end, we can precede the formatting with a "<" (left justify) or ">" (right justify). We demonstrate this with the following examples:

Output:

```
'Spam & Eggs:           6.99'
```

Output:

```
'        Spam & Eggs:   6.99'
```

Flag	Meaning
#	Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively.
0	The conversion result will be zero padded for numeric values.
–	The converted value is left adjusted
	If no sign (minus sign e.g.) is going to be written, a blank space is inserted before the value.
+	A sign character ("+" or "−") will precede the conversion (overrides a "space" flag).

Option	Meaning
'<'	The field will be left-aligned within the available space. This is usually the default for strings.
'>'	The field will be right-aligned within the available space. This is the default for numbers.

Option	Meaning

'0'	If the width field is preceded by a zero ('0') character, sign-aware zero-padding for numeric types will be enabled.
	`>>> x = 378`
	`>>> print("The value is {:06d}".format(x))`
	`The value is 000378`
	`>>> x = -378`
	`>>> print("The value is {:06d}".format(x))`
	`The value is -00378`

','	This option signals the use of a comma for a thousands separator.
	`>>> print("The value is {:,}".format(x))`
	`The value is 78,962,324,245`
	`>>> print("The value is {0:6,d}".format(x))`
	`The value is 5,897,653,423`
	`>>> x = 5897653423.89676`
	`>>> print("The value is {0:12,.3f}".format(x))`
	`The value is 5,897,653,423.897`

'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types.
'^'	Forces the field to be centered within the available space.

Unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

Additionally, we can modify the formatting with the sign option, which is only valid for number types:

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers, which is the default behavior.
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

USING DICTIONARIES IN "FORMAT"

We have seen in the previous chapters that we have two ways to access the values to be formatted:

Using the position or the index:

```
print("The capital of {0:s} is {1:s}".format("Ontario","Toronto"))
```

```
The capital of Ontario is Toronto
```

Just to mention it once more: We could have used empty curly braces in the previous example! Using keyword parameters:

```
print("The capital of {province} is {capital}".format(province="Ont
ario",capital="Toronto"))
```

```
The capital of Ontario is Toronto
```

The second case can be expressed with a dictionary as well, as we can see in the following code:

```
data = dict(province="Ontario",capital="Toronto")
data
```

Output:

```
{'province': 'Ontario', 'capital': 'Toronto'}
```

```
print("The capital of {province} is {capital}".format(**data))
```

```
The capital of Ontario is Toronto
```

The double "*" in front of data turns data automatically into the form 'province="Ontario",capital="Toronto"'. Let's look at the following Python program:

```
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]))
```

```
Countries and their capitals:
United States: Washington
US: Washington
Canada: Ottawa
Germany: Berlin
France: Paris
England: London
UK: London
Switzerland: Bern
Austria: Vienna
Netherlands: Amsterdam
```

We can rewrite the previous example by using the dictionary directly. The output will be the same:

```
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
                   "Austria" : "Vienna",
                   "Netherlands" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    format_string = c + ": {" + c + "}"
    print(format_string.format(**capital_country))
```

```
Countries and their capitals:
United States: Washington
US: Washington
Canada: Ottawa
Germany: Berlin
France: Paris
England: London
UK: London
Switzerland: Bern
Austria: Vienna
Netherlands: Amsterdam
```

USING LOCAL VARIABLE NAMES IN "FORMAT"

"locals" is a function, which returns a dictionary with the current scope's local variables, i.e- the local variable names are the keys of this dictionary and the corresponding values are the values of these variables:

```
a = 42
b = 47
def f():
    return 42
locals()
```

The dictionary returned by locals() can be used as a parameter of the string format method. This way we can use all the local variable names inside of a format string.

Continuing with the previous example, we can create the following output, in which we use the local variables a, b and f:

```
print("a={a}, b={b} and f={f}".format(**locals()))
```

```
a=42, b=47 and f=<function f at 0x00000285FFA729D8>
```

OTHER STRING METHODS FOR FORMATTING

The string class contains further methods, which can be used for formatting purposes as well: ljust, rjust, center and zfill.

Let S be a string, the 4 methods are defined like this:

- **center(...):**

```
S.center(width[, fillchar]) -> str
```

Return S centred in a string of length width. Padding is done using the specified fill character. The default value is a space.

Examples:

```
s = "Python"
s.center(10)
```

Output:

```
'  Python  '
```

```
s.center(10,"*")
```

Output:

```
'Price: $   356.09'
```

- **ljust(...):**

```
S.ljust(width[, fillchar]) -> str
```

Return S left-justified in a string of length "width". Padding is done using the specified fill character. If none is given, a space will be used as default.

Examples:

```
s = "Training"
s.ljust(12)
```

Output:

```
'Training    '
```

```
s.ljust(12,":")
```

Output:

```
'Training:::'
```

- **rjust(...):**

```
S.rjust(width[, fillchar]) -> str
```

Return S right-justified in a string of length width. Padding is done using the specified fill character. The default value is again a space.

Examples:

```
s = "Programming"
s.rjust(15)
```

Output:

```
'    Programming'
```

```
s.rjust(15, "~")
```

Output:

```
'~~~~Programming'
```

- **zfill(...):**

```
S.zfill(width) -> str
```

Pad a string S with zeros on the left, to fill a field of the specified width. The string S is never truncated. This method can be easily emulated with rjust.

Examples:

```
account_number = "43447879"
account_number.zfill(12)
# can be emulated with rjust:
```

Output:

```
'000043447879'
```

```
account_number.rjust(12,"0")
```

Output:

```
'000043447879'
```

FORMATTED STRING LITERALS

Python 3.6 introduces formatted string literals. They are prefixed with an 'f'. The formatting syntax is similar to the format strings accepted by str.format(). Like the format string of format method, they contain replacement fields formed with curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol. It's easiest to understand by looking at the following examples:

```
price = 11.23
f"Price in Euro: {price}"
```

Output:

```
'Price in Euro: 11.23'
```

```
f"Price in Swiss Franks: {price * 1.086}"
```

Output:

```
'Price in Swiss Franks: 12.195780000000001'
```

```
f"Price in Swiss Franks: {price * 1.086:5.2f}"
```

Output:

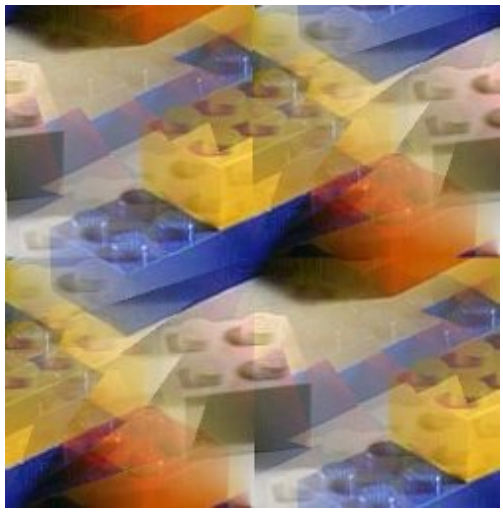
```
'Price in Swiss Franks: 12.20'
```

```
for article in ["bread", "butter", "tea"]:
    print(f"{article:>10}:")
```

```
     bread:
    butter:
       tea:
```

MODULAR PROGRAMMING AND MODULES

MODULAR PROGRAMMING



Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.

There is hardly any product nowadays, which doesn't heavily rely on modularisation, like cars and mobile phones. Computers belong to those products which are modularised to the utmost. So, what's a must for the hardware is an unavoidable necessity for the software running on the computers.

If you want to develop programs which are readable, reliable and maintainable without too much effort, you have to use some kind of modular software design. Especially if your application has a certain size. There exists a variety of concepts to design software in modular form. Modular programming is a software design technique to split your code into separate parts. These parts are called modules. The focus for this separation should be to have modules with no or just few dependencies upon other modules. In other words: Minimization of dependencies is the goal. When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

IMPORTING MODULES

So far we haven't explained what a Python module is. In a nutshell: every file, which has the file extension .py and consists of proper Python code, can be seen or is a module! There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:

In [ ]:

```
import math
```

The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:

```
math.pi
```

Output:

```
3.141592653589793
```

```
math.sin(math.pi/2)
```

Output:

```
1.0
```

```
math.cos(math.pi/2)
```

Output:

```
6.123233995736766e-17
```

```
math.cos(math.pi)
```

Output:

```
-1.0
```

It's possible to import more than one module in one import statement. In this case the module names are separated by commas:

In [ ]:

```
import math, random
```

import statements can be positioned anywhere in the program, but it's good style to place them directly at the beginning of a program.

If only certain objects of a module are needed, we can import only those:

In [ ]:

```
from math import sin, pi
```

The other objects, e.g. cos, are not available after this import. We are capable of accessing sin and pi directly, i.e. without prefixing them with "math." Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:

```
from math import *
sin(3.01) + tan(cos(2.1)) + e
```

Output:

```
2.2968833711382604
```

```
e
```

Output:

```
2.718281828459045
```

It's not recommended to use the asterisk notation in an import statement, except when working in the interactive Python shell. One reason is that the origin of a name can be quite obscure, because it can't be seen from which module it might have been imported. We will demonstrate another serious complication in the following example:

```
from numpy import *
from math import *
print(sin(3))
```

```
0.1411200080598672
```

```
sin(3)
```

Output:

```
0.1411200080598672
```

Let's modify the previous example slightly by changing the order of the imports:

```
from math import *
from numpy import *
print(sin(3))
```

```
0.1411200080598672
```

```
sin(3)
```

Output:

```
0.1411200080598672
```

People use the asterisk notation, because it is so convenient. It means avoiding a lot of tedious typing. Another way to shrink the typing effort lies in renaming a namespace. A good example for this is the numpy module. You will hardly find an example or a tutorial, in which they will import this module with the statement.

```
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])
```

Output:

```
array([[ 3,  0,  0,  0],
       [ 0, 11,  0,  0],
       [ 0,  0,  7,  0],
       [ 0,  0,  0,  9]])
```

DESIGNING AND WRITING MODULES

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done, we just save the following code in the file fibonacci.py:

```
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)
```

Output:

```
13
```

```
fibonacci.fib(20)
```

Output:

```
6765
```

```
fibonacci.ifib(42)
```

Output:

```
267914296
```

```
fibonacci.ifib(1000)
```

Output:

```
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75
```

Don't try to call the recursive version of the Fibonacci function with large arguments like we did with the iterative version. A value like 42 is already too large. You will have to wait for a long time!

As you can easily imagine: It's a pain if you have to use those functions often in your program and you always have to type in the fully qualified name, i.e. fibonacci.fib(7). One solution consists in assigning a local name to a module function to get a shorter name:

```
fib = fibonacci.ifib
fib(10)
```

Output:

```
55
```

But it's better, if you import the necessary functions directly into your module, as we will demonstrate further down in this chapter.

MORE ON MODULES

Usually, modules contain functions or classes, but there can be "plain" statements in them as well. These statements can be used to initialize the module. They are only executed when the module is imported.

Let's look at a module, which only consists of just one statement:

```
print("The module is imported now!")
```

```
The module is imported now!
```

We save with the name "one_time.py" and import it two times in an interactive session:

```
import one_time
```

```
The module is imported now!
```

We can see that it was only imported once. Each module can only be imported once per interpreter session or in a program or script. If you change a module and if you want to reload it, you must restart the interpreter again. In Python 2.x, it was possible to reimport the module by using the built-in reload, i.e.reload(modulename):

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more informatio
n.
import one_time
The module is imported now!
reload(one_time)
The module is imported now!
```

This is not possible anymore in Python 3.x. You will cause the following error:

```
import one_time
```

```
reload(one_time)

---------------------------------------------------------------
---------------
NameError                                 Traceback (most r
ecent call last)
<ipython-input-7-102e1bec2702> in <module>
----> 1 reload(one_time)

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

Since Python 3.0 the reload built-in function has been moved into the imp standard library module. So it's still possible to reload files as before, but the functionality has to be imported. You have to execute an "import imp" and use imp.reload(my_module). Alternatively, you can use "imp import reload" and use reload(my_module).

Example with reloading the Python3 way:

```
$ python3
Python 3.1.2 (r312:79147, Sep 27 2010, 09:57:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more informatio
n.
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)
```

Output:

```
139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125
```

This does not introduce the module name from which the imports are taken in the local symbol table. It's possible but not recommended to import all names defined in a module, except those beginning with an underscore "_":

In [ ]:

```
from fibonacci import *
fib(500)
```

This shouldn't be done in scripts but it's possible to use it in interactive sessions to save typing.

EXECUTING MODULES AS SCRIPTS

Essentially a Python module is a script, so it can be run as a script:

```
python fibo.py
```

The module which has been started as a script will be executed as if it had been imported, but with one exception: The system variable **name** is set to "**main**". So it's possible to program different behaviour into a module for the two cases. With the following conditional statement the file can be used as a module or as a script, but only if it is run as a script the method fib will be started with a command line argument:

```
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)
```

Output:

```
8.0
```

```
sinus(pi)
```

Output:

```
1.2246467991473532e-16
```

KINDS OF MODULES

There are different kind of modules:

- Those written in Python
  They have the suffix: .py
- Dynamically linked C modules
  Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter
  It's possible to get a complete list of these modules:

```
import sys
print(sys.builtin_module_names)
```

```
('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_
cn', '_codecs_hk', '_codecs_iso2022', '_codecs_jp', '_codec
s_kr', '_codecs_tw', '_collections', '_contextvars', '_cs
v', '_datetime', '_functools', '_heapq', '_imp', '_io', '_j
son', '_locale', '_lsprof', '_md5', '_multibytecodec', '_op
code', '_operator', '_pickle', '_random', '_sha1', '_sha25
6', '_sha3', '_sha512', '_signal', '_sre', '_stat', '_strin
g', '_struct', '_symtable', '_thread', '_tracemalloc', '_wa
rnings', '_weakref', '_winapi', 'array', 'atexit', 'audioo
p', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandle
r', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt',
'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipi
mport', 'zlib')
```

An error message is returned for Built-in-Modules.

MODULE SEARCH PATH

If you import a module, let's say "import xyz", the interpreter searches for this module in the following locations and in the order given:

```
The directory of the top-level file, i.e. the file being executed.
The directories of PYTHONPATH, if this global environment variable of
your operating system is set.
standard installation path Linux/Unix e.g. in /usr/lib/python3.5.
```

It's possible to find out where a module is located after it has been imported:

```
import numpy
numpy.file
'/usr/lib/python3/dist-packages/numpy/init.py'



import random
random.file
'/usr/lib/python3.5/random.py'


</pre>
The file attribute doesn't always exist. This is the case with module
s which are statically linked C libraries.
```

```
import math
math.__file__
```

```
---------------------------------------------------------------
----------------
AttributeError                            Traceback (most r
ecent call last)
<ipython-input-4-bb98ec32d2a8> in <module>
      1 import math
----> 2 math.__file__

AttributeError: module 'math' has no attribute '__file__'
```

CONTENT OF A MODULE

With the built-in function dir() and the name of the module as an argument, you can list all valid attributes and methods for that module.

```
import math
dir(math)
```

Output:

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

Calling dir() without an argument, a list with the names in the current local scope is returned:

```
import math
cities = ["New York", "Toronto", "Berlin", "Washington", "Amsterda
m", "Hamburg"]
dir()
```

Output:

```
['In',
 'Out',
 '_',
 '_1',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'builtins',
 'cities',
 'exit',
 'get_ipython',
 'math',
 'quit']
```

It's possible to get a list of the Built-in functions, exceptions, and other objects by importing the builtins module:

```
import builtins
dir(builtins)
```

Output:

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
 'SyntaxWarning',
 'SystemError',
 'SystemExit',
```

```
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
```

```
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

PACKAGES

If you have created a lot of modules at some point in time, you may loose the overview about them. You may have dozens or hundreds of modules and they can be categorized into different categories. It is similar to the situation in a file system: Instead of having all files in just one directory, you put them into different ones, being organized according to the topics of the files. We will show in the next chapter of our Python tutorial how to organize modules into packages.

Class 1 exercises

Exercise 1.1 The cover price of a book is $24.95, but bookstores get a 40 percent discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. Calculate the total wholesale costs for 60 copies.

Exercise 1.2 Can you identify and explain the errors in the following lines of code? Correct them.

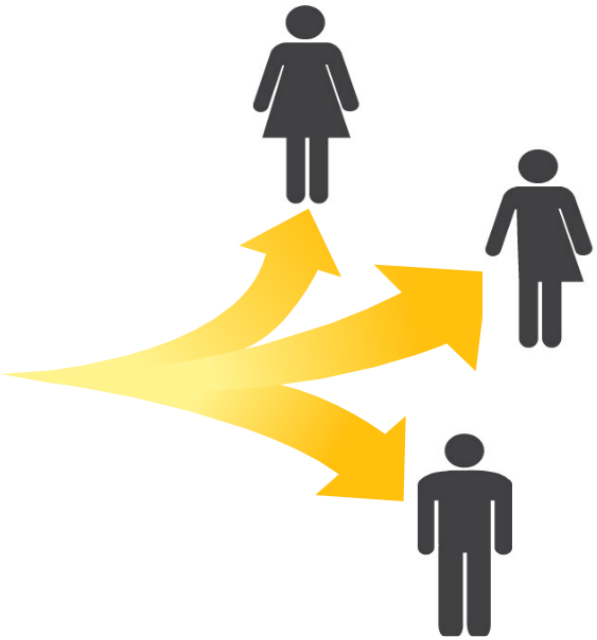
Exercise 1.3 Define three variables var1, var2 and var3. Calculate the average of these variables and assign it to average. Print the average. Add three comments.

Exercise 1.4 Write code that can compute the surface of circle, using the variables radius and pi = 3.14159. The formula, in case you do not know, is radius times radius times pi. Print the outcome of your program as follows: “The surface area of a circle with radius ... is ...”

Class 2
Conditions and iterations

CONDITIONAL STATEMENTS

DECISION MAKING



Do you belong to those people who find it hard to make decisions? In real life, I mean. In certain circumstances, some decisions in daily life are unavoidable, e.g. people have to go their separate ways from time to time, as our picture shows. We all have to make decisions all the time, ranging from trivial issues like choosing what to wear in the morning, what to have for breakfast, right up to life-changing decisions like taking or changing a job, deciding what to study, and many more. However, there you are, having decided to learn more about the conditional statements in Python.

Conditions - usually in the form of if statements - are one of the key features of a programming language, and Python is no exception. You will hardly find a programming language without an if statement.[1] There is hardly a way to program without branches in the code flow, at least if the code needs to solve a useful problem.

A decision must be made when the script or program comes to a point where there is a selection of actions, i.e. different calculations from which to choose.

The decision, in most cases, depends on the value of variables or arithmetic expressions. These expressions are evaluated using the Boolean True or False values. The instructions for decision making are called conditional statements. In a programming language, therefore, the parts of the conditional statements are executed under the given conditions. In many cases there are two code parts: One which will be executed, if the condition is True, and another one, if it is False. In other words, a branch determines which of two (or even more) program parts (alternatives) will be executed depending on one (or more) conditions. Conditional statements and branches belong to the control structures of programming languages, because with their help a program can react to different states that result from inputs and calculations.

CONDITIONAL STATEMENTS IN REAL LIFE

The principle of the indentation style is also known in natural languages, as we can deduce from the following text:

```
 If it rains tomorrow, I'll clean up the basemen
t.
 After that, I will paint the walls. If there is
 any time left, I will file my tax return.
```

Of course, as we all know, there will be no time left to file the tax return. Jokes aside: in the previous text, as you see, we have a sequence of actions that must be performed in chronological order. If you take a closer look at the text, you will find some ambiguity in it: is 'the painting of the walls' also related to the event of the rain? Will the tax declaration be done with or without rain? What will this person do, if it does not rain? Therefore, we extend the text with additional alternatives:

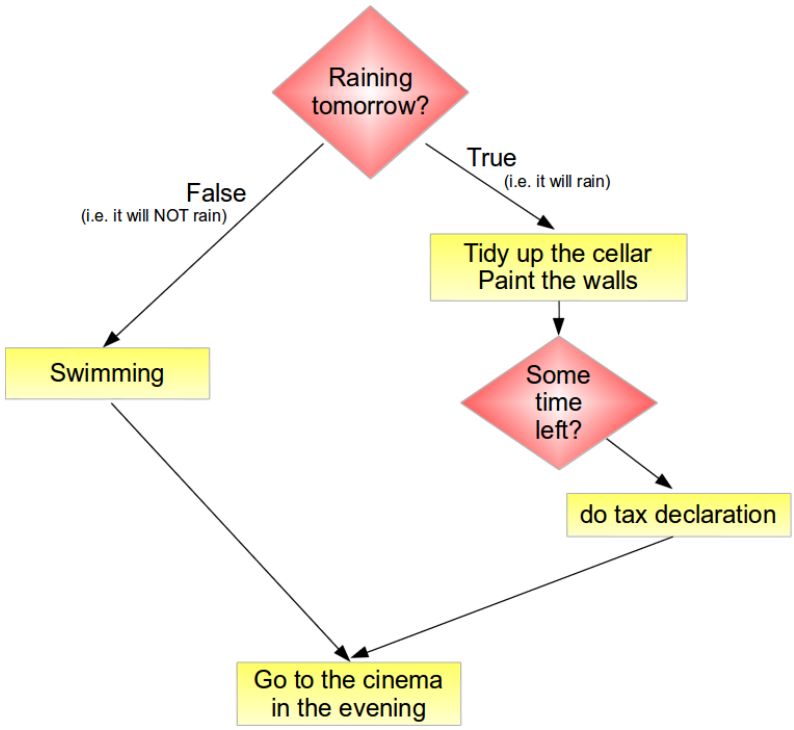
```
If it rains tomorrow, I'll clean up the basement.
After that I will paint the walls. If there is
any time left, I will file my tax return. Otherwise,
i.e. if it will not rain, I will go swimming.
```

```
In the evening, I will go to the cinema with my wife!
```

It is still nor clear. Can his wife hope to be invited to the cinema? Does she have to pray or hope for rain? To make the text clear, we can phrase it to be closer to the programming code and the Python coding style, and hopefully there will be a happy ending for his wife:

```
If it rains tomorrow, I will do the following:
     - Clean up the basement
     - Paint the walls
     - If there is any time left, I will make my
       tax declaration
Otherwise I will do the following:
     - Go swimming
Going to the movies in the evening with my wife
```

We can also graphically depict this. Such a workflow is often referred to in the programming environment as a so-called flowchart or program schedule (PAP):



To encode conditional statements in Python, we need to know how to combine statements into a block. So this seems to be the ideal moment to introduce the Python Block Principle in combination with the conditional statements.

COMBINING STATEMENTS INTO BLOCKS

In programming, blocks are used to treat sets of statements as if they were a single statement. A block consists of one or more statements. A program can be considered a block consisting of statements and other nested blocks. In programming languages, there exist various approaches to syntactically describing blocks: ALGOL 60 and Pascal, for example, use "begin" and "end", while C and similar languages use curly braces "{" and "}". Bash has another design that uses 'do ... done' and 'if ... fi' or 'case ... esac' constructs. All these languages have in common is that the indentation style is not binding. The problems that may arise from this will be shown in the next subchapter. You can safely skip this, if you are only interested in Python. For inexperienced programmers, this chapter may seem a bit difficult.

EXCURSION TO C

For all these approaches, there is a big drawback: The code may be fine for the interpreter or compiler of the language, but it can be written in a way that is poorly structured for humans. We want to illustrate this in the following "pseudo" C-like code snippet:

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

We will add a couple of spaces in front of calling 'go_cinema'. This puts this call at the same level as 'enjoy_swimming':

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

The execution of the program will not change: you will go to the cinema, whether it will be raining or not! However, the way the code is written falsely suggests that you will only go to the cinema if it does not rain. This example shows the possible ambiguity in the interpretation of C code by humans. In other words, you can inadvertently write the code, which leads to a procedure that does not "behave" as intended.
In this example lurks even another danger. What if the programmer had just forgotten to put the last statement out of the curly braces?

Should the code look like this:

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

In the following program you will only go to the cinema, if it is not a rainy day. That only makes sense if it's an open-air cinema.

The following code is the right code to go to the movies regardless of the weather:

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

The problem with C is that the way the code is indented has no meaning for the compiler, but it can lead to misunderstanding, if misinterpreted.

This is different in Python. Blocks are created with indentations, in other words, blocks are based on indentation. We can say "What you see is what you get!"
The example above might look like this in a Python program:

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

There is no ambiguity in the Python version. The cinema visit of the couple will take place in any case, regardless of the weather. Moving the go_cinema() call to the same indentation level as enjoy_swimming() immediately changes the program logic. In that case, there will be no cinema if it rains.

We have just seen that it is useful to use indentations in C or C++ programs to increase the readability of a program. For the compilation or execution of a program they are but irrelevant. The compiler relies exclusively on the structuring determined by the brackets. Code in Python, on the other hand, can only be structured with indents. That is, Python forces the programmer to use indentation that should be used anyway to write nice and readable codes. Python does not allow you to obscure the structure of a program by misleading indentations.

Each line of a code block in Python must be indented with the same number of spaces or tabs. We will continue to delve into this in the next subsection on conditional statements in Python.

CONDITIONAL STATEMENTS IN PYTHON

The if statement is used to control the program flow in a Python program. This makes it possible to decide at runtime whether certain program parts should be executed or not.
The simplest form of an if statement in a Python program looks like this:

```
if condition:
    statement
    statement
    # further statements, if necessary
```

The indented block is only executed if the condition 'condition' is True. That is, if it is logically true.
The following example asks users for their nationality. The indented block will only be executed if the nationality is "french". If the user enters another nationality, nothing happens.

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

```
Nationality? french
Préférez-vous parler français?
```

Please note that if "French" is entered in the above program, nothing will happen. The check in the condition is case-sensitive and in this case only responds when "french" is entered in lower case. We now extend the condition with an "or"

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

```
Nationality? french
Préférez-vous parler français?
```

The Italian colleagues might now feel disadvantaged because Italian speakers were not considered. We extend the program by another if statement.

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

```
Nationality? italian
Preferisci parlare italiano?
```

This little script has a drawback. Suppose that the user enters "french" as a nationality. In this case, the block of the first "if" will be executed. Afterwards, the program checks if the second "if" can be executed as well. This doesn't make sense, because we know that the input "french" does not match the condition "italian" or "Italian". In other words: if the first 'if' matched, the second one can never match as well. In the end, this is an unnecessary check when entering "french" or "French".

We solve the problem with an "elif" condition. The 'elif' expression is only checked if the expression of a previous "elif" or "if" was false.

```
person = input("Nationality? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
elif person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
else:
    print("You are neither French nor Italian.")
    print("So, let us speak English!")
```

```
Nationality? german
You are neither French nor Italian.
So, let us speak English!
```

As in our example, most if statements also have "elif" and "else" branches. This means, there may be more than one "elif" branches, but only one "else" branch. The "else" branch must be at the end of the if statement. Other "elif" branches can not be attached after an 'else'. 'if' statements don't need either 'else' nor 'elif' statements.

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

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2

...

elif another_condition:
    another_statement_block
else:
    else_block
```

If the condition "condition_1" is True, the statements of the block statement"_block_1" will be executed. If not, condition_2 will be evaluated. If "condition_2" evaluates to True, statement"_block_2" will be executed, if condition_2 is False, the other conditions of the following elif conditions will be checked, and finally if none of them has been evaluated to True, the indented block below the else keyword will be executed.

EXAMPLE WITH IF CONDITION

For children and dog lovers it is an interesting and frequently asked question how old their dog would be if it was not a dog, but a human being. To calculate this task, there are various scientific and pseudo-scientific approaches. An easy approach can be:

```
 A one-year-old dog is roughly equivalent to a 14-year-old human bein
g
 A dog that is two years old corresponds in development to a 22 year
 old person.
 Each additional dog year is equivalent to five human years.
```

The following example program in Python requires the age of the dog as the input and calculates the age in human years according to the above rule. 'input' is a statement, where the program flow stops and waits for user input printing out the message in brackets:

```
age = int(input("Age of the dog: "))
print()
if age < 0:
    print("This cannot be true!")
elif age == 0:
    print("This corresponds to 0 human years!")
elif age == 1:
    print("Roughly 14 years!")
elif age == 2:
    print("Approximately 22 years!")
else:
    human = 22 + (age -2) * 5
    print("Corresponds to  " + str(human) + " human years!")
```

```
Age of the dog: 1

Roughly 14 years!
```

In [ ]:

```
We will use this example again in our tutorial
```

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

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

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

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

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

```
1st Number: 1
2nd Number: 2
3rd Number: 3
The maximal value is: 3.0
```

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

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

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

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

```
1st Number: 4.4
2nd Number: 1
3rd Number: 8.3
The maximal value is: 8.3
```

Another way to find the maximum can be seen in the following example. We are using the built-in function max, which calculates the maximum of a list or a tuple of numerical values:

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

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

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

```
1st Number: 1
2nd Number: 4
3rd Number: 5.5
The maximal value is: 5.5
```

TERNARY IF STATEMENT

Let's look at the following English sentence, which is valid in Germany :

The maximum speed is 50 if we are within the city, otherwise it is 100.

This can be translated into ternary Python statements:

```
inside_city_limits = True
maximum_speed = 50 if inside_city_limits else 100
print(maximum_speed)
```

```
50
```

This code looks like an abbreviation for the following code:

```
im_ort = True
if im_ort:
    maximale_geschwindigkeit = 50
else:
    maximale_geschwindigkeit = 100
print(maximale_geschwindigkeit)
```

```
50
```

But it is something fundamentally different. "50 if inside_city_limits else 100" is an expression that we can also use in function calls.

EXERCISES

EXERCISE 1

- A leap year is a calendar year containing an additional day added to keep the calendar year synchronized with the astronomical or seasonal year. In the Gregorian calendar, each leap year has 366 days instead of 365, by extending February to 29 days rather than the common 28. These extra days occur in years which are multiples of four (with the exception of centennial years not divisible by 400). Write a Python program, which asks for a year and calculates, if this year is a leap year or not.

EXERCISE 2

- Body mass index (BMI) is a value derived from the mass (weight) and height of a person. The BMI is defined as the body mass divided by the square of the body height, and is universally expressed in units of kg/m2, resulting from mass in kilograms and height in metres. $BMI = \frac{m}{l^2}$ Write a program, which asks for the length and the weight of a person and returns an evaluation string according to the following table:

Category	BMI (kg/m^2)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

SOLUTIONS

EXERCISE 1:

We will use the modulo operator in the following solution. 'n % m' returns the remainder, if you divide (integer division) n by m. '5 % 3' returns 2 for example.

```
year = int(input("Which year? "))

if year % 4:
    # not divisible by 4
    print("no leap year")
elif year % 100 == 0 and year % 400 != 0:
    print("no leap year")
else:
    print("leap year")
```

```
Which year? 2020
leap year
```

EXERCISE 2:

```
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

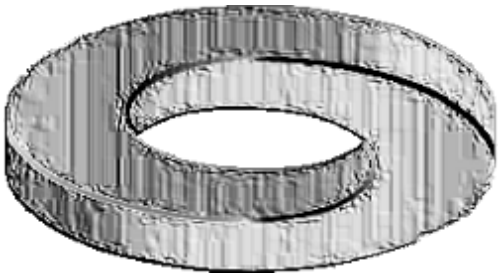
```
What is your height? 180
What is your weight? 74
0.002283950617283951
Very severely underweight
```

FOOTNOTES

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

FOR LOOPS

INTRODUCTION



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

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

Different kinds of for loops:

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

  This is by far the most common type. This statement is the one used by C. The header of this kind of for loop consists of a three-parameter loop control expression. Generally it has the form: for (A; Z; I) A is the initialisation part, Z determines a termination expression and I is the counting expression, where the loop variable is incremented or dcremented. An example of this kind of loop is the for-loop of the programming language C: for (i=0; i <= n; i++) This kind of for loop is not implemented in Python!

- Numeric Ranges

  This kind of for loop is a simplification of the previous kind. It's a counting or enumerating loop. Starting with a start value and counting up to an end value, like for i = 1 to 100 Python doesn't use this either.

- Vectorized for loops

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

- Iterator-based for loop

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

SYNTAX OF THE FOR LOOP

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

for in : else:

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

Example of a simple for loop in Python:

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

```
C
C++
Perl
Python
```

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

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

```
edibles = ["bacon", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious bacon
No more spam please!
Finally, I finished stuffing myself
```

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

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

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

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

print("Finally, I finished stuffing myself")
```

```
Great, delicious bacon
No more spam please!
Great, delicious eggs
Great, delicious nuts
Finally, I finished stuffing myself
```

THE RANGE() FUNCTION

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

```
range(5)
```

Output:

```
range(0, 5)
```

This result is not self-explanatory. It is an object which is capable of producing the numbers from 0 to 4. We can use it in a for loop and you will see what is meant by this:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

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

```
list(range(10))
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range() can also be called with two arguments:

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

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

Example:

```
range(4, 10)
```

Output:

```
range(4, 10)
```

```
list(range(4, 10))
```

Output:

```
[4, 5, 6, 7, 8, 9]
```

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

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

Example with step:

```
list(range(4, 50, 5))
```

Output:

```
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```

It can be done backwards as well:

```
list(range(42, -12, -7))
```

Output:

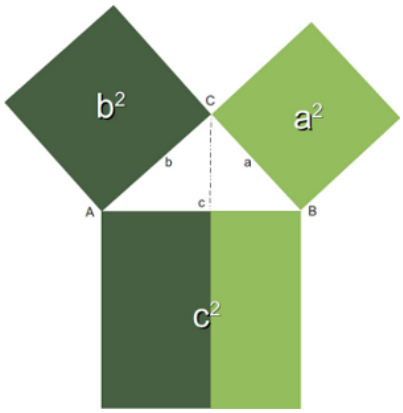
```
[42, 35, 28, 21, 14, 7, 0, -7]
```

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

```
n = 100

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

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

```
Sum of 1 until 100: 5050
```

CALCULATION OF THE PYTHAGOREAN NUMBERS



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

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

So the definition is very simple: Three integers satisfying a2+b2=c2 are called Pythagorean numbers.

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

In [ ]:

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

ITERATING OVER LISTS WITH RANGE()

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

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

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

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

LIST ITERATION WITH SIDE EFFECTS

If you loop over a list, it's best to avoid changing the list in the loop body. Take a look at the following example:

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

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

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

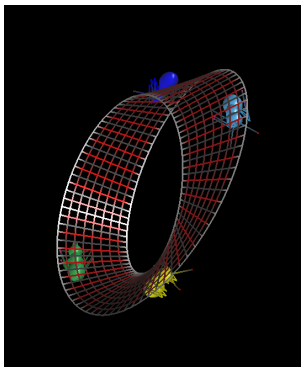
```
colours = ["red"]
for i in colours[:]:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)
```

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

We still might have done something, we shouldn't have done. We changed the list "colours", but our change didn't have any effect on the loop. The elements to be looped remained the same during the iterations.

LOOPS

GENERAL STRUCTURE OF A LOOP



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

Essentially, there are three different kinds of loops:

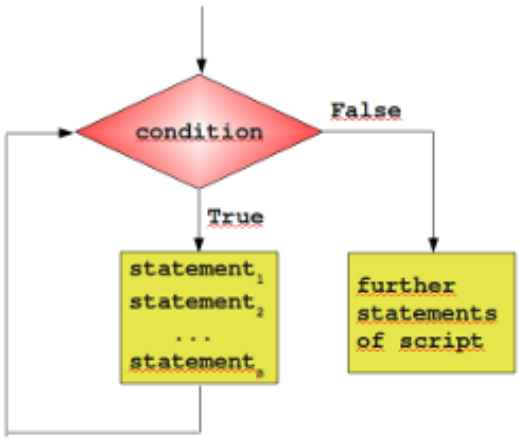
- Count-controlled loops A construction for repeating a loop a certain number of times. An example of this kind of loop is the for-loop of the programming language C:

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

  Python doesn't have this kind of loop.

- Condition-controlled loop A loop will be repeated until a given condition changes, i.e. changes from True to False or from False to True, depending on the kind of loop. There are 'while loops' and 'do while' loops with this behaviour.

- Collection-controlled loop This is a special construct which allows looping through the elements of a 'collection', which can be an array, list or other ordered sequence. Like the for loop of the bash shell (e.g. for i in *, do echo $i; done) or the foreach loop of Perl.



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

Most loops contain a counter or more generally, variables, which change their values in the course of calculation. These variables have to be initialized before the loop is started. The counter or other variables, which can be altered in the body of the loop, are contained in the condition. Before the body of the loop is executed, the condition is evaluated. If it evaluates to False, the while loop is finished. In other words, the program flow will continue with the first statement after the while statement, i.e. on the same indentation level as the while loop. If the condition is evaluated to True, the body, - the indented block below the line with "while" - gets executed. After the body is finished, the condition will be evaluated again. The body of the loop will be executed as long as the condition yields True.

A SIMPLE EXAMPLE WITH A WHILE LOOP

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

```
n = 100

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

print("Sum of 1 until " + str(n) +  " results in " + str(total_sum
))
```

```
Sum of 1 until 100 results in 5050
```

EXERCISE

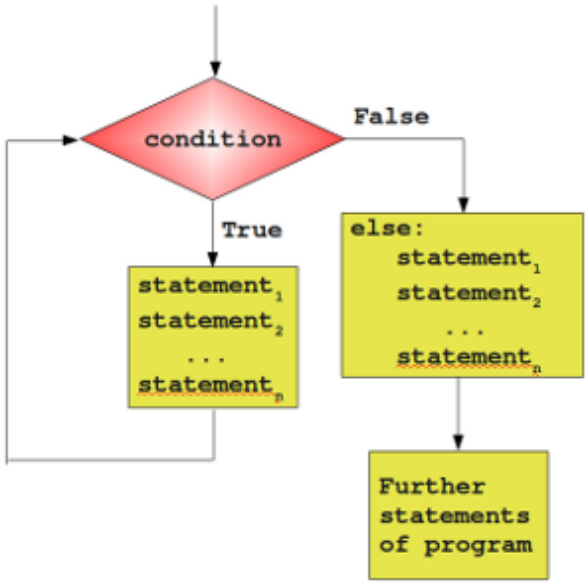
Write a program, which asks for the initial balance K0 and for the interest rate. The program shall calculate the new capital K1 after one year including the interest.

Extend the program with a while-loop, so that the capital Kn after a period of n years can be calculated.

```
K = float(input("Starting capital? "))
p = float(input("Interest rate? "))
n = int(input("Number of years? "))

i = 0
while i < n:
    K +=  K * p / 100.0
    # or K *=  1 +  p / 100.0
    i += 1
    print(i, K)
print("Capital after " + str(n) + " ys: " + str(K))
```

```
1 100.6
2 101.2036
3 101.8108216
4 102.4216865296
5 103.0362166487776
6 103.65443394867026
7 104.27636055236228
8 104.90201871567645
9 105.53143082797051
10 106.16461941293834
Capital after 10 ys: 106.16461941293834
```

THE ELSE PART



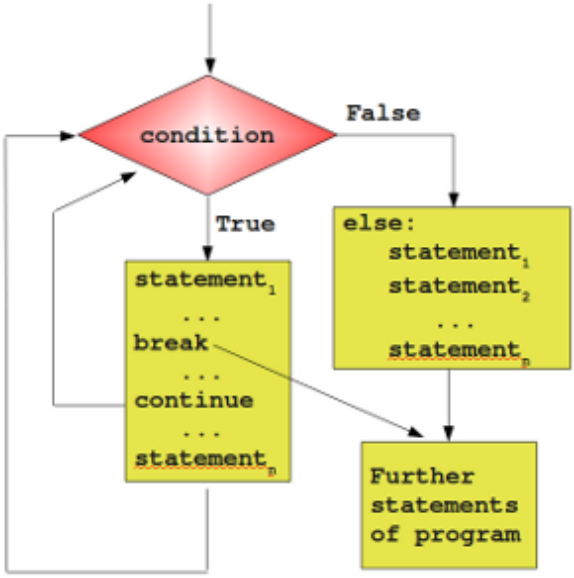
Similar to the if statement, the while loop of Python has also an optional else part. This is an unfamiliar construct for many programmers of traditional programming languages. The statements in the else part are executed, when the condition is not fulfilled anymore. Some might ask themselves now, where the possible benefit of this extra branch is. If the statements of the additional else part were placed right after the while loop without an else, they would have been executed anyway, wouldn't they? We need to understand a new language construction, i.e. the break statement, to obtain a benefit from the additional else branch of the while loop. The general syntax of a while loop looks like this:

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

```
 while potatoes:
    peel

    cut
else:
    pot pot on oven
    cook for 40 minutes

wash salad
and so on
```

PREMATURE TERMINATION OF A WHILE LOOP

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



This behaviour will be illustrated in the following example, a little guessing number game. A human player has to guess a number between a range of 1 to n. The player inputs his guess. The program informs the player, if this number is larger, smaller or equal to the secret number, i.e. the number which the program has randomly created. If the player wants to gives up, he or she can input a 0 or a negative number. Hint: The program needs to create a random number. Therefore it's necessary to include the module "random".

```
import random
upper_bound = 20
lower_bound = 1
#to_be_guessed = int(n * random.random()) + 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess > 0:
        if guess > to_be_guessed:
            print("Number too large")
        elif guess < to_be_guessed:
            print("Number too small")
    else:
        # 0 means giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
else:
    print("Congratulations. You made it!")
```

```
Number too small

Number too large

Number too large

Number too large

Congratulations. You made it!
```

The previous program doesn't check if the `number` makes sense, i.e. if the number is between the boundaries `upper_bound` and `over_bound` . We can improve our program:

```
import random
upper_bound = 20
lower_bound = 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:    # giving up
        print("Sorry that you're giving up!")
        break    # break out of a loop, don't execute "else"
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess > to_be_guessed:
        print("Number too large")
    elif guess < to_be_guessed:
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
Congratulations. You made it!
```

The boundaries should be adapted according to the user input:

```
import random
upper_bound = 20
lower_bound = 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:    # giving up
        print("Sorry that you're giving up!")
        break    # break out of a loop, don't execute "else"
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess > to_be_guessed:
        upper_bound = guess - 1
        print("Number too large")
    elif guess < to_be_guessed:
        lower_bound = guess + 1
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

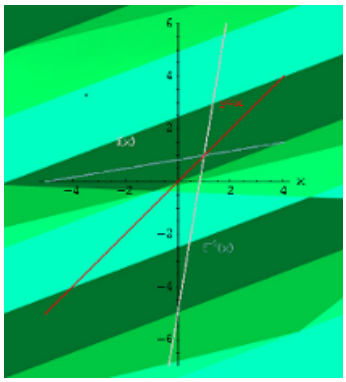
```
Number too small

Number too small

Number too large

Number too small

Congratulations. You made it!
```

```
import random
upper_bound = 20
lower_bound = 1
adaptive_upper_bound = upper_bound
adaptive_lower_bound = lower_bound
#to_be_guessed = int(n * random.random()) + 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess == 0:   # giving up
        print("Sorry that you're giving up!")
        break   # break out of a loop, don't execute "else"
    if guess < lower_bound or guess > upper_bound:
        print("guess not within boundaries!")
    elif guess < adaptive_lower_bound or guess > adaptive_upper_bou
nd:
        print("Your guess is contradictory to your previous guesse
s!")
    elif guess > to_be_guessed:
        adaptive_upper_bound = guess - 1
        print("Number too large")
    elif guess < to_be_guessed:
        adaptive_lower_bound = guess + 1
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
Number too small

Your guess is contradictory to your previous guesses!

Number too small

Your guess is contradictory to your previous guesses!

guess not within boundaries!

Number too small

Number too small

Number too small

Number too small

Congratulations. You made it!
```

FUNCTIONS

SYNTAX



The concept of a function is one of the most important in mathematics. A common usage of functions in computer languages is to implement mathematical functions. Such a function is computing one or more results, which are entirely determined by the parameters passed to it.

This is mathematics, but we are talking about programming and Python. So what is a function in programming? In the most general sense, a function is a structuring element in programming languages to group a bunch of statements so they can be utilized in a program more than once. The only way to accomplish this without functions would be to reuse code by copying it and adapting it to different contexts, which would be a bad idea. Redundant code - repeating code in this case - should be avoided! Using functions usually enhances the comprehensibility and quality of a program. It also lowers the cost for development and maintenance of the software.

Functions are known under various names in programming languages, e.g. as subroutines, routines, procedures, methods, or subprograms.

MOTIVATING EXAMPLE OF FUNCTIONS

Let us look at the following code:

```
print("Program starts")

print("Hi Peter")
print("Nice to see you again!")
print("Enjoy our video!")

# some lines of codes
the_answer = 42

print("Hi Sarah")
print("Nice to see you again!")
print("Enjoy our video!")

width, length = 3, 4
area = width * length

print("Hi Dominque")
print("Nice to see you again!")
print("Enjoy our video!")
```

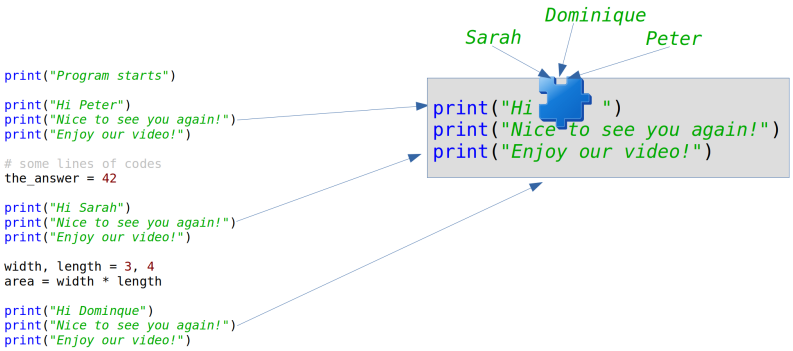
```
Program starts
Hi Peter
Nice to see you again!
Enjoy our video!
Hi Sarah
Nice to see you again!
Enjoy our video!
Hi Dominque
Nice to see you again!
Enjoy our video!
```

Let us have a closer look at the code above:

```
print("Program starts")

print("Hi Peter")
print("Nice to see you again!")
print("Enjoy our video!")

# some lines of codes
the_answer = 42

print("Hi Sarah")
print("Nice to see you again!")
print("Enjoy our video!")

just_some_code = "whatever"
another_var = "whatever you do"

print("Hi Dominque")
print("Nice to see you again!")
print("Enjoy our video!")
```

You can see in the code that we are greeting three persons. Every time we use three print calls which are nearly the same. Just the name is different. This is what we call redundant code. We are repeating the code the times. This shouldn't be the case. This is the point where functions can and should be used in Python.

We could use wildcards in the code instead of names. In the following diagram we use a piece of the puzzle. We only write the code once and replace the puzzle piece with the corresponding name:



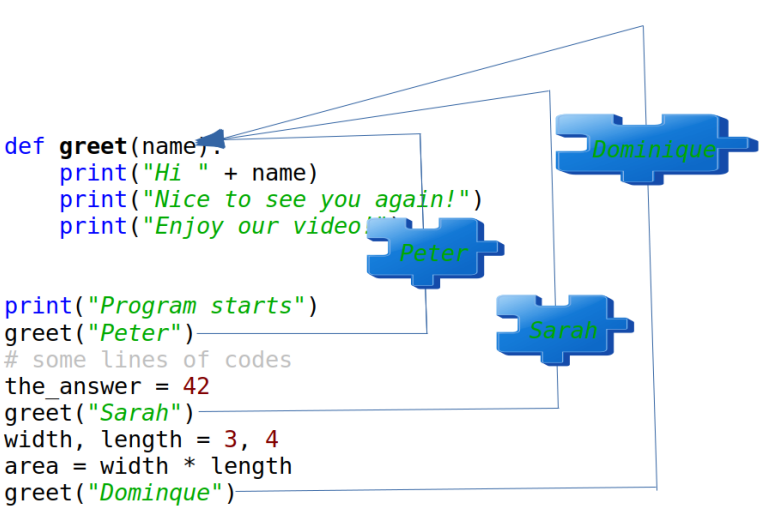
Of course, this was no correct Python code. We show how to do this in Python in the following section.

FUNCTIONS IN PYTHON

The bollowing code uses a function with the name `greet` . The previous puzzle piece is now a parameter with the name "name":

```
def greet(name):
    print("Hi " + name)
    print("Nice to see you again!")
    print("Enjoy our video!")


print("Program starts")

greet("Peter")


# some lines of codes
the_answer = 42

greet("Sarah")

width, length = 3, 4
area = width * length


greet("Dominque")
```

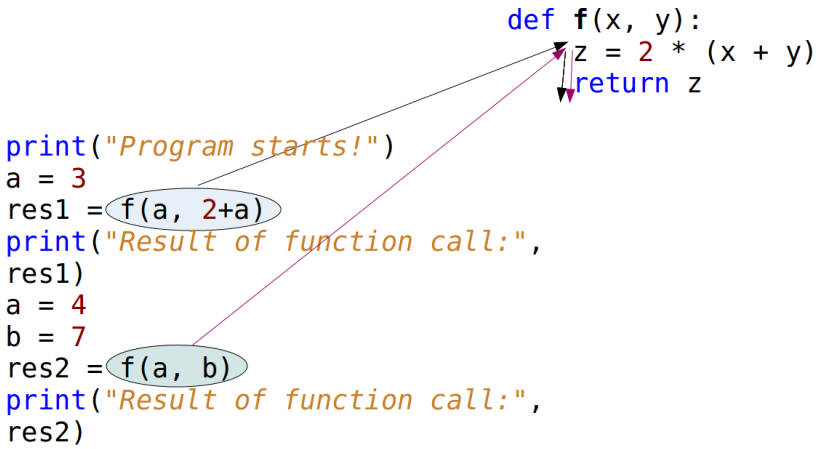
```
Program starts
Hi Peter
Nice to see you again!
Enjoy our video!
Hi Sarah
Nice to see you again!
Enjoy our video!
Hi Dominque
Nice to see you again!
Enjoy our video!
```



We used a function in the previous code. We saw that a function definition starts with the `def` keyword. The general syntax looks like this:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called. We demonstrate this in the following picture:



The code from the picture can be seen in the following:

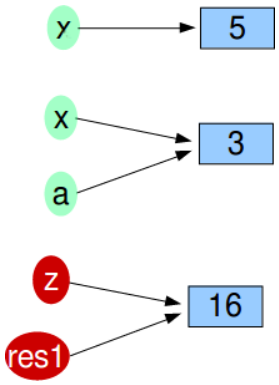
```
def f(x, y):
    z = 2 * (x + y)
    return z


print("Program starts!")
a = 3
res1 = f(a, 2+a)
print("Result of function call:", res1)
a = 4
b = 7
res2 = f(a, b)
print("Result of function call:", res2)
```

```
Program starts!
Result of function call: 16
Result of function call: 22
```

We call the function twice in the program. The function has two parameters, which are called `x` and `y`. This means that the function `f` is expecting two values, or I should say "two objects". Firstly, we call this function with `f(a, 2+a)`. This means that `a` goes to `x` and the result of `2+a` (5) 'goes to' the variable `y`. The mechanism for assigning arguments to parameters is called **argument passing**. When we reach the `return` statement, the object referenced by `z` will be return, which means that it will be assigned to the variable `res1`. After leaving the function `f`, the variable `z` and the parameters `x` and `y` will be deleted automatically.

```
def f(x, y):
    z = 2 * (x + y)
    return z

print("Program starts!")
a = 3
res1 = f(a, 2+a)
print("Result of function call:",
res1)
a = 4
b = 7
res2 = f(a, b)
print("Result of function call:",
res2)
```

The references to the objects can be seen in the next diagram:



The next Python code block contains an example of a function without a return statement. We use the `pass` statement inside of this function. `pass` is a null operation. This means that when it is executed, nothing happens. It is useful as a placeholder in situations when a statement is required syntactically, but no code needs to be executed:

```
def doNothing():
    pass
```

A more useful function:

```
def fahrenheit(T_in_celsius):
    """ returns the temperature in degrees Fahrenheit """
    return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
    print(t, ": ", fahrenheit(t))
```

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

""" returns the temperature in degrees Fahrenheit """ is the so-called docstring. It is used by the help function:

```
help(fahrenheit)

Help on function fahrenheit in module __main__:

fahrenheit(T_in_celsius)
    returns the temperature in degrees Fahrenheit
```

Our next example could be interesting for the calorie-conscious Python learners. We also had an exercise in the chapter Conditional Statements of our Python tutorial. We will create now a function version of the program.

The body mass index (BMI) is a value derived from the mass (w) and height (l) of a person. The BMI is defined as the body mass divided by the square of the body height, and is universally expressed

$$BMI = \frac{w}{l^2}$$

The weight is given in kg and the length in metres.

```
def BMI(weight, height):
    """ calculates the BMI where
        weight is in kg and height in metres"""
    return weight / height**2
```

We like to write a function to evaluate the bmi values according to the following table:

Category	BMI (kg/m²)		BMI Prime	
	from	to	from	to
Very severely underweight		15		0.60
Severely underweight	15	16	0.60	0.64
Underweight	16	18.5	0.64	0.74
Normal (healthy weight)	18.5	25	0.74	1.0
Overweight	25	30	1.0	1.2
Obese Class I (Moderately obese)	30	35	1.2	1.4
Obese Class II (Severely obese)	35	40	1.4	1.6
Obese Class III (Very severely obese)	40	45	1.6	1.8
Obese Class IV (Morbidly Obese)	45	50	1.8	2
Obese Class V (Super Obese)	50	60	2	2.4
Obese Class VI (Hyper Obese)	60		2.4	

```
height = float(input("What is your height? "))
weight = float(input("What is your weight? "))

bmi = weight / height ** 2
print(bmi)
if bmi < 15:
    print("Very severely underweight")
elif bmi < 16:
    print("Severely underweight")
elif bmi < 18.5:
    print("Underweight")
elif bmi < 25:
    print("Normal (healthy weight)")
elif bmi < 30:
    print("Overweight")
elif bmi < 35:
    print("Obese Class I (Moderately obese)")
elif bmi < 40:
    print("Obese Class II (Severely obese)")
else:
    print("Obese Class III (Very severely obese)")
```

```
23.69576446280992
Normal (healthy weight)
```

DEFAULT ARGUMENTS IN PYTHON

When we define a Python function, we can set a default value to a parameter. If the function is called without the argument, this default value will be assigned to the parameter. This makes a parameter optional. To say it in other words: Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

We will demonstrate the operating principle of default parameters with a simple example. The following function `hello` , - which isn't very useful, - greets a person. If no name is given, it will greet everybody:

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

hello("Peter")
hello()
```

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

THE DEFAULTS PITFALL

In the previous section we learned about default parameters. Default parameters are quite simple, but quite often programmers new to Python encounter a horrible and completely unexpected surprise. This surprise arises from the way Python treats the default arguments and the effects steming from mutable objects.

Mutable objects are those which can be changed after creation. In Python, dictionaries are examples of mutable objects. Passing mutable lists or dictionaries as default arguments to a function can have unforeseen effects. Programmer who use lists or dictionaries as default arguments to a function, expect the program to create a new list or dictionary every time that the function is called. However, this is not what actually happens. Default values will not be created when a function is called. Default values are created exactly once, when the function is defined, i.e. at compile-time.

Let us look at the following Python function "spammer" which is capable of creating a "bag" full of spam:

```
def spammer(bag=[]):
    bag.append("spam")
    return bag
```

Calling this function once without an argument, returns the expected result:

```
spammer()
```

Output:

```
['spam']
```

The surprise shows when we call the function again without an argument:

```
spammer()
```

Output:

```
['spam', 'spam']
```

Most programmers will have expected the same result as in the first call, i.e. `['spam']`

To understand what is going on, you have to know what happens when the function is defined. The compiler creates an attribute `__defaults__` :

```
def spammer(bag=[]):
    bag.append("spam")
    return bag

spammer.__defaults__
```

Output:

```
([],)
```

Whenever we will call the function, the parameter `bag` will be assigned to the list object referenced by `spammer.__defaults__[0]` :

```
for i in range(5):
    print(spammer())

print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam', 'spam']
['spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam', 'spam']
spammer.__defaults__ (['spam', 'spam', 'spam', 'spam', 'spa
m'],)
```

Now, you know and understand what is going on, but you may ask yourself how to overcome this problem. The solution consists in using the immutable value `None` as the default. This way, the function can set bag dynamically (at run-time) to an empty list:

```
def spammer(bag=None):
    if bag is None:
        bag = []
    bag.append("spam")
    return bag

for i in range(5):
    print(spammer())

print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam']
['spam']
['spam']
['spam']
spammer.__defaults__ (None,)
```

DOCSTRING

The first statement in the body of a function is usually a string statement called a Docstring, which can be accessed with the `function_name.__doc__` . For example:

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

print("The docstring of the function hello: " + hello.__doc__)
```

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

KEYWORD PARAMETERS

Using keyword parameters is an alternative way to make function calls. The definition of the function doesn't change. An example:

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

print(sumsub(12, 4))
print(sumsub(42, 15, d=10))
```

```
8
17
```

Keyword parameters can only be those, which are not used as positional arguments. We can see the benefit in the example. If we hadn't had keyword parameters, the second call to function would have needed all four arguments, even though the c argument needs just the default value:

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

```
17
```

RETURN VALUES

In our previous examples, we used a return statement in the function sumsub but not in Hello. So, we can see that it is not mandatory to have a return statement. But what will be returned, if we don't explicitly give a return statement. Let's see:

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

res = no_return(4, 5)
print(res)
```

```
None
```

If we start this little script, *None* will be printed, i.e. the special value *None* will be returned by a return-less function. *None* will also be returned, if we have just a return in a function without an expression:

```
def empty_return(x, y):
    c = x + y
    return

res = empty_return(4, 5)
print(res)
```

```
None
```

Otherwise the value of the expression following return will be returned. In the next example 9 will be printed:

```
def return_sum(x, y):
    c = x + y
    return c

res = return_sum(4, 5)
print(res)
```

```
9
```

Let's summarize this behavior: Function bodies can contain one or more return statements. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value `None` is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value `None` will be returned.

RETURNING MULTIPLE VALUES

A function can return exactly one value, or we should better say one object. An object can be a numerical value, like an integer or a float. But it can also be e.g. a list or a dictionary. So, if we have to return, for example, 3 integer values, we can return a list or a tuple with these three integer values. That is, we can indirectly return multiple values. The following example, which is calculating the Fibonacci boundary for a positive number, returns a 2-tuple. The first element is the Largest Fibonacci Number smaller than x and the second component is the Smallest Fibonacci Number larger than x. The return value is immediately stored via unpacking into the variables lub and sup:

```
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 = 0, 1
    while True:
        if new < x:
            old, new = new, old+new
        else:
            if new == x:
                new = old + new
            return (old, new)

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

```
Largest Fibonacci Number smaller than x: 4181
Smallest Fibonacci Number larger than x: 6765
```

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()
```

```
Python
```

```
def f():
    s = "Perl"
    print(s)
f()
```

```
Perl
```

```
s = "Python"
f()
print(s)
```

```
Perl
Python
```

```
def f():
    print(s)
    s = "Perl"
    print(s)


s = "Python"
f()
print(s)
```

```
---------------------------------------------------------------
----------------
UnboundLocalError                         Traceback (most r
ecent call last)
<ipython-input-25-81b2fbbc4d42> in <module>
      6
      7 s = "Python"
----> 8 f()
      9 print(s)

<ipython-input-25-81b2fbbc4d42> in f()
      1 def f():
----> 2     print(s)
      3     s = "Perl"
      4     print(s)
      5

UnboundLocalError: local variable 's' referenced before ass
ignment
```

If we execute the previous script, we get the error message: UnboundLocalError: local variable 's' referenced before assignment.

The variable s is ambigious in f(), i.e. in the first print in f() the global s could be used with the value "Python". After this we define a local variable s with the assignment s = "Perl".

```
def f():
    global s
    print(s)
    s = "dog"
    print(s)
s = "cat"
f()
print(s)
```

```
cat
dog
dog
```

We made the variable s global inside of the script. Therefore anything we do to s inside of the function body of f is done to the global variable s outside of f.

ARBITRARY NUMBER OF PARAMETERS

There are many situations in programming, in which the exact number of necessary parameters cannot be determined a-priori. An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple reference. This asterisk shouldn't be mistaken for the C syntax, where this notation is connected with pointers. Example:

```
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))
```

```
61.0
42502.3275
38.5
45.0
```

This is great, but we have still have one problem. You may have a list of numerical values. Like, for example,

```
x = [3, 5, 9]
```

You cannot call it with

```
arithmetic_mean(x)
```

because "arithmetic_mean" can't cope with a list. Calling it with

```
arithmetic_mean(x[0], x[1], x[2])
```

Output:

```
5.666666666666667
```

is cumbersome and above all impossible inside of a program, because list can be of arbitrary length.

The solution is easy: The star operator. We add a star in front of the x, when we call the function.

```
arithmetic_mean(*x)
```

Output:

```
5.666666666666667
```

This will "unpack" or singularize the list.

A practical example for `zip` and the star or asterisk operator: We have a list of 4, 2-tuple elements:

```
my_list = [('a', 232),
           ('b', 343),
           ('c', 543),
           ('d', 23)]
```

We want to turn this list into the following 2 element, 4-tuple list:

```
[('a', 'b', 'c', 'd'),
(232, 343, 543, 23)]
```

This can be done by using the *-operator and the zip function in the following way:

```
list(zip(*my_list))
```

Output:

```
[('a', 'b', 'c', 'd'), (232, 343, 543, 23)]
```

ARBITRARY NUMBER OF KEYWORD PARAMETERS

In the previous chapter we demonstrated how to pass an arbitrary number of positional parameters to a function. It is also possible to pass an arbitrary number of keyword parameters to a function as a dictionary. To this purpose, we have to use the double asterisk "**"

```
def f(**kwargs):
    print(kwargs)

f()
```

```
{}
```

```
f(de="German",en="English",fr="French")
```

```
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

One use case is the following:

```
def f(a, b, x, y):
    print(a, b, x, y)
d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

EXERCISES WITH FUNCTIONS

EXERCISE 1

Rewrite the "dog age" exercise from chapter Conditional Statements as a function.

The rules are:

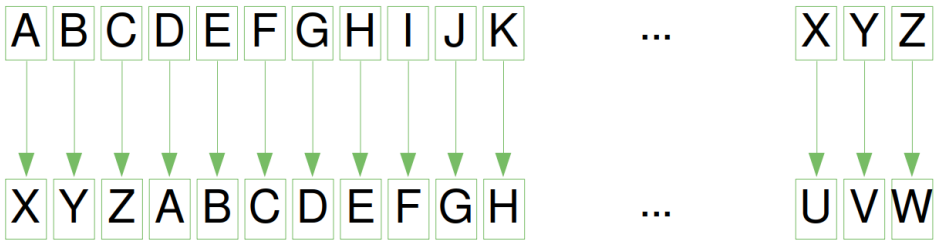
- A one-year-old dog is roughly equivalent to a 14-year-old human being
- A dog that is two years old corresponds in development to a 22 year old person.
- Each additional dog year is equivalent to five human years.

EXERCISE 2

Write a function which takes a text and encrypts it with a Caesar cipher. This is one of the simplest and most commonly known encryption techniques. Each letter in the text is replaced by a letter some fixed number of positions further in the alphabet.

What about decrypting the coded text?

The Caesar cipher is a substitution cipher.



EXERCISE 3

We can create another substitution cipher by permutating the alphet and map the letters to the corresponding permutated alphabet.

Write a function which takes a text and a dictionary to decrypt or encrypt the given text with a permutated alphabet.

EXERCISE 4

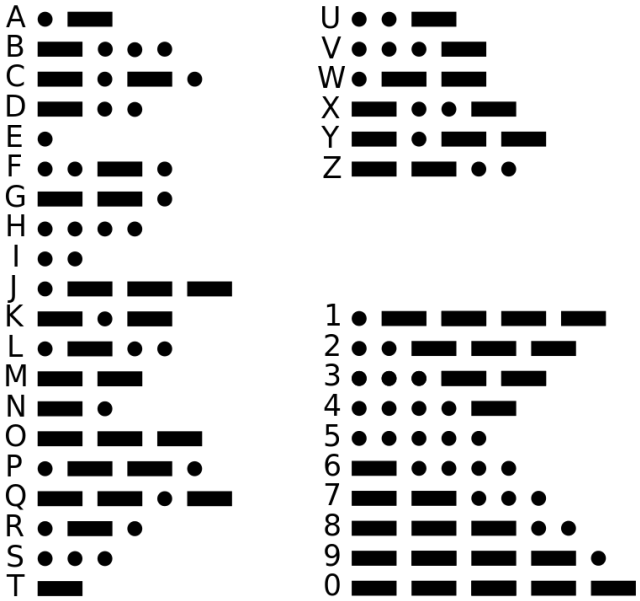
Write a function txt2morse, which translates a text to morse code, i.e. the function returns a string with the morse code.

Write another function morse2txt which translates a string in Morse code into a „normal“ string.

The Morse character are separated by spaces. Words by three spaces.

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.



EXERCISE 5

Perhaps the first algorithm used for approximating $\sqrt{S}$ is known as the "Babylonian method", named after the Babylonians, or "Hero's method", named after the first-century Greek mathematician Hero of Alexandria who gave the first explicit description of the method.

If a number x_n is close to the square root of a then

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

will be a better approximation.

Write a program to calculate the square root of a number by using the Babylonian method.

EXERCISE 6

Write a function which calculates the position of the n-th occurence of a string sub in another string s. If sub doesn't occur in s, -1 shall be returned.

EXERCISE 7

Write a function `fuzzy_time` which expects a time string in the form hh: mm (e.g. "12:25", "04:56"). The function rounds up or down to a quarter of an hour. Examples:

fuzzy_time("12:58") ---> "13

SOLUTIONS

SOLUTION TO EXERCISE 1

```
def dog_age2human_age(dog_age):
    """dog_age2human_age(dog_age)"""
    if dog_age == 1:
        human_age = 14
    elif dog_age == 2:
        human_age = 22
    else:
        human_age =  22 + (dog_age-2)*5
    return human_age

age = int(input("How old is your dog? "))
print(f"This corresponds to {dog_age2human_age(age)} human years!")
```

```
This corresponds to 37 human years!
```

SOLUTION TO EXERCISE 2

```
import string
abc = string.ascii_uppercase
def caesar(txt, n, coded=False):
    """ returns the coded or decoded text """
    result = ""
    for char in txt.upper():
        if char not in abc:
            result += char
        elif coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hello, here I am!", n)
print(x)
print(caesar(x, n, True))
```

```
EBIIL, EBOB F XJ!
HELLO, HERE I AM!
```

In the previous solution we only replace the letters. Every special character is left untouched. The following solution adds some special characters which will be also permutated. Special charcters not in abc will be lost in this solution!

```
import string
abc = string.ascii_uppercase + " .,-?!"
def caesar(txt, n, coded=False):
    """ returns the coded or decoded text """
    result = ""
    for char in txt.upper():
        if coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hello, here I am!", n)
print(x)
print(caesar(x, n, True))

x = caesar("abcdefghijkl", n)
print(x)
```

```
EBIILZXEBOBXFX-J,
HELLO, HERE I AM!
-?!ABCDEFGHI
```

We will present another way to do it in the following implementation. The advantage is that there will be no calculations like in the previous versions. We do only lookups in the decrypted list 'abc_cipher':

```
import string

shift = 3
abc = string.ascii_uppercase + " .,-?!"
abc_cipher = abc[-shift:] + abc[:-shift]
print(abc_cipher)

def caesar (txt, shift):
    """Encodes the text "txt" to caesar by shifting it 'shift' posi
tions """
    new_txt=""
    for char in txt.upper():
        position = abc.find(char)
        new_txt +=abc_cipher[position]
    return new_txt

n = 3
x = caesar("Hello, here I am!", n)
print(x)

x = caesar("abcdefghijk", n)
print(x)
```

```
-?!ABCDEFGHIJKLMNOPQRSTUVWXYZ .,
EBIILZXEBOBXFX-J,
-?!ABCDEFGH
```

SOLUTION TO EXERCISE 3

```
import string
from random import sample

alphabet = string.ascii_letters
permutated_alphabet = sample(alphabet, len(alphabet))

encrypt_dict = dict(zip(alphabet, permutated_alphabet))
decrypt_dict = dict(zip(permutated_alphabet, alphabet))

def encrypt(text, edict):
    """ Every character of the text 'text'
    is mapped to the value of edict. Characters
    which are not keys of edict will not change"""
    res = ""
    for char in text:
        res = res + edict.get(char, char)
    return res

# Donald Trump: 5:19 PM, September 9 2014
txt = """Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional 'global warming' is worse."""

ctext = encrypt(txt, encrypt_dict)
print(ctext + "\n")
print(encrypt(ctext, decrypt_dict))
```

```
OQlerQGGk yDd xVd nDdyxdkx
xVDdyx Ql xVd Fz xo hoxV hyGe
yle noGedl dynGdk. gdeQy HGyQrk
EQHxQolyG 'nGohyG MyDrQln' Qk MoDkd.

Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional 'global warming' is worse.
```

Alternative solution:

```
import string
alphabet = string.ascii_lowercase + "äöüß .?!\n"

def caesar_code(alphabet, c, text, mode="encoding"):
    text = text.lower()
    coded_alphabet = alphabet[-c:] + alphabet[0:-c]
    if mode == "encoding":
        encoding_dict = dict(zip(alphabet, coded_alphabet))
    elif mode == "decoding":
        encoding_dict = dict(zip(coded_alphabet, alphabet))
    print(encoding_dict)
    result = ""
    for char in text:
        result += encoding_dict.get(char, "")
    return result

txt2 = caesar_code(alphabet, 5, txt)
caesar_code(alphabet, 5, txt2, mode="decoding")
```

```
{'a': ' ', 'b': '.', 'c': '?', 'd': '!', 'e': '\n', 'f':
'a', 'g': 'b', 'h': 'c', 'i': 'd', 'j': 'e', 'k': 'f', 'l':
'g', 'm': 'h', 'n': 'i', 'o': 'j', 'p': 'k', 'q': 'l', 'r':
'm', 's': 'n', 't': 'o', 'u': 'p', 'v': 'q', 'w': 'r', 'x':
's', 'y': 't', 'z': 'u', 'ä': 'v', 'ö': 'w', 'ü': 'x', 'ß':
'y', ' ': 'z', '.': 'ä', '?': 'ö', '!': 'ü', '\n': 'ß'}
{' ': 'a', '.': 'b', '?': 'c', '!': 'd', '\n': 'e', 'a':
'f', 'b': 'g', 'c': 'h', 'd': 'i', 'e': 'j', 'f': 'k', 'g':
'l', 'h': 'm', 'i': 'n', 'j': 'o', 'k': 'p', 'l': 'q', 'm':
'r', 'n': 's', 'o': 't', 'p': 'u', 'q': 'v', 'r': 'w', 's':
'x', 't': 'y', 'u': 'z', 'v': 'ä', 'w': 'ö', 'x': 'ü', 'y':
'ß', 'z': ' ', 'ä': '.', 'ö': '?', 'ü': '!', 'ß': '\n'}
```

Output:

```
'windmills are the greatest \nthreat in the us to both bald
\nand golden eagles. media claims \nfictional global warmin
g is worse.'
```

SOLUTION TO EXERCISE 4

```
latin2morse_dict = {'A':'.-', 'B':'-...', 'C':'-.-.', 'D':'-..',
                    'E':'.', 'F':'..-.', 'G':'--.','H':'....',
                    'I':'..', 'J':'.---', 'K':'-.-', 'L':'.-..',
                    'M':'--', 'N':'-.', 'O':'---', 'P':'.--.',
                    'Q':'--.-', 'R':'.-.', 'S':'...', 'T':'-',
                    'U':'..-', 'V':'...-', 'W':'.--', 'X':'-..-',
                    'Y':'-.--', 'Z':'--..', '1':'.----', '2':'..--
',
                    '3':'...--', '4':'....-', '5':'.....', '6':
'-....',
                    '7':'--...', '8':'---..', '9':'----.', '0':'---
--',
                    ',':'--..--', '.':'.-.-.-', '?':'..--..', ';':
'-.-.-',
                    ':':'---...', '/':'-..-.', '-':'-....-', '\'':
'.----.',
                    '(':'-.--.-', ')':'-.--.-', '[':'-.--.-', ']':
'-.--.-',
                    '{':'-.--.-', '}':'-.--.-', '_':'..--.-'}

# reversing the dictionary:
morse2latin_dict = dict(zip(latin2morse_dict.values(),
                            latin2morse_dict.keys()))

print(morse2latin_dict)
```

```
{'.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E',
'..-.': 'F', '--.': 'G', '....': 'H', '..': 'I', '.---':
'J', '-.-': 'K', '.-..': 'L', '--': 'M', '-.': 'N', '---':
'O', '.--.': 'P', '--.-': 'Q', '.-.': 'R', '...': 'S', '-':
'T', '..-': 'U', '...-': 'V', '.--': 'W', '-..-': 'X', '-.-
-': 'Y', '--..': 'Z', '.----': '1', '...--': '3', '....-':
'4', '.....': '5', '-....': '6', '--...': '7', '---..':
'8', '----.': '9', '-----': '0', '--..--': ',', '.-.-.-':
'.', '..--..': '?', '-.-.-': ';', '---...': ':', '-..-.':
'/', '-....-': '-', '.----.': "'", '-.--.-': '}', '..--.-':
'_'}
```

```
def txt2morse(txt, alphabet):
    morse_code = ""
    for char in txt.upper():
        if char == " ":
            morse_code += "   "
        else:
            morse_code += alphabet[char] + " "
    return morse_code

def morse2txt(txt, alphabet):
    res = ""
    mwords = txt.split("   ")
    for mword in mwords:
        for mchar in mword.split():
            res += alphabet[mchar]
        res += " "
    return res

mstring = txt2morse("So what?", latin2morse_dict)
print(mstring)
print(morse2txt(mstring, morse2latin_dict))
```

```
... ---    .-- .... .- - ..--..
SO WHAT?
```

SOLUTION TO EXERCISE 5

```
def heron(a, eps=0.000000001):
    """ Approximate the square root of a"""
    previous = 0
    new = 1
    while abs(new - previous) > eps:
        previous = new
        new = (previous + a/previous) / 2
    return new

print(heron(2))
print(heron(2, 0.001))
```

```
1.414213562373095
1.4142135623746899
```

SOLUTION TO EXERCISE 6

```
def findnth(s, sub, n):
    num = 0
    start = -1
    while num < n:
        start = s.find(sub, start+1)
        if start == -1:
            break
        num += 1

    return start

s = "abc xyz abc jkjkjk abc lkjkjlkj abc jlj"
print(findnth(s,"abc", 3))
```

```
19
```

```python
def vage_uhrzeit(zeit):
    stunden, minuten = zeit.split(":")
    minuten = int(minuten)
    stunden = int(stunden)
    minuten = minuten / 60   # Wandlung in Dezimalminuten
    # Werte in 0.0, 0.25, 0.5, 0.75:
    minuten = round(minuten * 4, 0) / 4
    if minuten == 0:
        print("Ungefähr " + str(stunden) + " Uhr!")
    elif minuten == 0.25:
        print("Ungefähr viertel nach " + str(stunden) + "!")
    else:
        if stunden == 12:
            stunden = 1
        else:
            stunden += 1
        if minuten == 0.50:
            print("Es ist ungefähr halb " + str(stunden) + "!")
        elif minuten == 0.75:
            print("Ungefähr eine viertel vor " + str(stunden) + "!"
)
        elif minuten == 1.00:
            print("Ungefähr " + str(stunden) + " Uhr!")

stunde = "12"
for x in range(0, 60, 3):
    vage_uhrzeit(stunde +":" + f"{x:02d}")
```

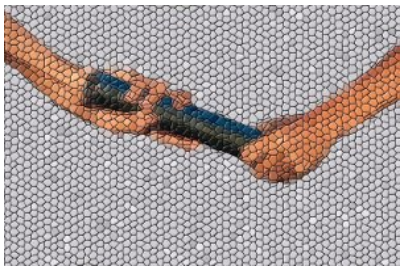
```
Ungefähr 12 Uhr!
Ungefähr 12 Uhr!
Ungefähr 12 Uhr!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Ungefähr viertel nach 12!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Es ist ungefähr halb 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr eine viertel vor 1!
Ungefähr 1 Uhr!
Ungefähr 1 Uhr!
```

PARAMETERS AND ARGUMENTS

A function or procedure usually needs some information about the environment, in which it has been called. The interface between the environment, from which the function has been called, and the function, i.e. the function body, consists of special variables, which are called parameters. By using these parameters, it's possible to use all kind of objects from "outside" inside of a function. The syntax for how parameters are declared and the semantics for how the arguments are passed to the parameters of the function or procedure depends on the programming language.

Very often the terms parameter and argument are used synonymously, but there is a clear difference. Parameters are inside functions or procedures, while arguments are used in procedure calls, i.e. the values passed to the function at run-time.

"CALL BY VALUE" AND "CALL BY NAME"



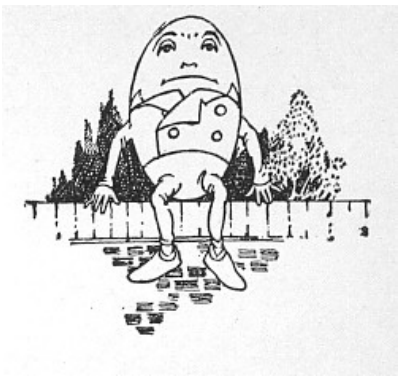
The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs between programming languages. The most common evaluation strategies are "call by value" and "call by reference":

- Call by Value The most common strategy is the call-by-value evaluation, sometimes also called pass-by-value. This strategy is used in C and C++, for example. In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter. This ensures that the variable in the caller's scope will stay unchanged when the function returns.

- Call by Reference In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed. By using Call by Reference we save both computation time and memory space, because arguments do not need to be copied. On the other hand this harbours the disadvantage that variables can be "accidentally" changed in a function call. So, special care has to be taken to "protect" the values, which shouldn't be changed. Many programming languages support call-by-reference, like C or C++, but Perl uses it as default.

In ALGOL 60 and COBOL there has been a different concept called call-by-name, which isn't used anymore in modern languages.

AND WHAT ABOUT PYTHON?



There are some books which call the strategy of Python call-by-value, and some call it call-by-reference. You may ask yourself, what is right.

Humpty Dumpty supplies the explanation:

--- "When I use a word," Humpty Dumpty said, in a rather a scornful tone, "it means just what I choose it to mean - neither more nor less."

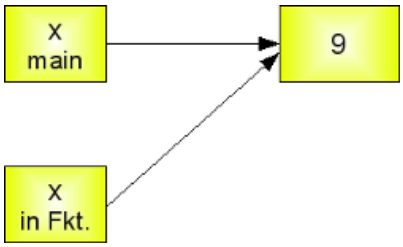
--- "The question is," said Alice, "whether you can make words mean so many different things."

--- "The question is," said Humpty Dumpty, "which is to be master - that's all."

Lewis Carroll, Through the Looking-Glass

To come back to our initial question what evaluation strategy is used in Python: the authors who call the mechanism call-by-value and those who call it call-by-reference are stretching the definitions until they fit.

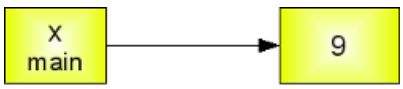
Correctly speaking, Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".



If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place within the function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

First, let's have a look at the integer variables below. The parameter inside the function remains a reference to the argument's variable, as long as the parameter is not changed. As soon as a new value is assigned to it, Python creates a separate local variable. The caller's variable will not be changed this way:

```
def ref_demo(x):
    print("x=",x," id=",id(x))
    x=42
    print("x=",x," id=",id(x))
```



In the example above, we used the id() function, which takes an object as a parameter. id(obj) returns the "identity" of the object "obj". This identity, the return value of the function, is an integer which is unique and constant for this object during its lifetime. Two different objects with non-overlapping lifetimes may have the same id() value.

If you call the function ref_demo() of the previous example - like we do in the green block further below - we can check what happens to x with the id() function: We can see that in the main scope, x has the identity 140709692940944. In the first print statement of the ref_demo() function, the x from the main scope is used, because we can see that we get the same identity. After we assigned the value 42 to x, x gets a new identity 140709692942000, i.e. a separate memory location from the global x. So, when we are back in the main scope x has still the original value 9 and the id 140709692940944.

In other words, Python initially behaves like call-by-reference, but as soon as we change the value of such a variable, i.e. as soon as we assign a new object to it, Python "switches" to call-by-value. That is, a local variable x will be created and the value of the global variable x will be copied into it.

```
x = 9
id(x)
```

Output:

```
140709692940944
```

```
ref_demo(x)
```

```
x= 9  id= 140709692940944
x= 42  id= 140709692942000
```

```
id(x)
```

Output:

```
140709692940944
```

SIDE EFFECTS

A function is said to have a side effect, if, in addition to producing a return value, it modifies the caller's environment in other ways. For example, a function might modify a global or static variable, modify one of its arguments, raise an exception, write data to a display or file etc.

There are situations, in which these side effects are intended, i.e. they are part of the function's specification. But in other cases, they are not wanted , they are hidden side effects. In this chapter, we are only interested in the side effects that change one or more global variables, which have been passed as arguments to a function. Let's assume, we are passing a list to a function. We expect the function not to change this list. First, let's have a look at a function which has no side effects. As a new list is assigned to the parameter list in func1(), a new memory location is created for list and list becomes a local variable.

```
def no_side_effects(cities):
    print(cities)
    cities = cities + ["Birmingham", "Bradford"]
    print(cities)
    locations = ["London", "Leeds", "Glasgow", "Sheffield"]
no_side_effects(locations)
```

```
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Birmi
ngham', '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 as "side_effects" in the following example:

```
def side_effects(cities):
    print(cities)
    cities += ["Birmingham", "Bradford"]
    print(cities)

locations = ["London", "Leeds", "Glasgow", "Sheffield"]
side_effects(locations)
```

```
['London', 'Leeds', 'Glasgow', 'Sheffield']
['London', 'Leeds', 'Glasgow', 'Sheffield', 'Birmingham',
'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[:])
print(locations)
```

```
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg', 'Pari
s', 'Marseille']
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
```

```
print(locations)
```

```
['Lyon', 'Toulouse', 'Nice', 'Nantes', 'Strasbourg']
```

We can see that the global list locations has not been effected by the execution of the function.

COMMAND LINE ARGUMENTS

If you use a command line interface, i.e. a text user interface (TUI) , and not a graphical user interface (GUI), command line arguments are very useful. They are arguments which are added after the function call in the same line.

It's easy to write Python scripts using command line arguments. If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces. Inside the script these arguments are accessible through the list variable sys.argv. The name of the script is included in this list sys.argv[0]. sys.argv[1] contains the first parameter, sys.argv[2] the second and so on. The following script (arguments.py) prints all arguments:

```
 # Module sys has to be imported:
import sys
```

ITERATION OVER ALL ARGUMENTS:

```
for eachArg in sys.argv:
        print(eachArg)
```

Example call to this script:

```
python argumente.py python course for beginners
```

This call creates the following output:

```
argumente.py
python
course
for
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")
locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux", "Mars
eille")
```

```
Paris ()
Paris ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseill
e')
```

* IN FUNCTION CALLS

A * can appear in function calls as well, as we have just seen in the previous exercise: The semantics is in this case "inverse" to a star in a function definition. An argument will be unpacked and not packed. In other words, the elements of the list or tuple are singularized:

```
def f(x,y,z):
    print(x,y,z)

p = (47,11,12)
f(*p)
```

```
47 11 12
```

There is hardly any need to mention that this way of calling our function is more comfortable than the following one:

```
f(p[0],p[1],p[2])
```

```
47 11 12
```

Additionally, the previous call (f(p[0],p[1],p[2])) doesn't work in the general case, i.e. lists of unknown lengths. "Unknown" means that the length is only known at runtime and not when we are writing the script.

ARBITRARY KEYWORD PARAMETERS

There is also a mechanism for an arbitrary number of keyword parameters. To do this, we use the double asterisk "**" notation:

```
>>> def f(**args):
    print(args)

f()
```

```
{}
```

```
f(de="German",en="English",fr="French")
```

```
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

DOUBLE ASTERISK IN FUNCTION CALLS

The following example demonstrates the usage of ** in a function call:

```
def f(a,b,x,y):
    print(a,b,x,y)

d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

and now in combination with *:

```
t = (47,11)
d = {'x':'extract','y':'yes'}
f(*t, **d)
```

```
47 11 extract yes
```

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. We wanted to enforce that we always have a non-empty list of numbers. This is necessary to prevent a division by zero error, because the average of an empty list of numbers is not defined.

In the following interactive Python session, we can learn how to use this function. We assume that the function arithmetic_mean is saved in a file called statistics.py.

```
from statistics import arithmetic_mean
arithmetic_mean(4,7,9)
6.666666666666667
arithmetic_mean(4,7,9,45,-3.7,99)
26.71666666666667
```

This works fine, but there is a catch. What if somebody wants to call the function with a list, instead of a variable number of numbers, as we have shown above? We can see in the following that we raise an error, as most hopefully, you might expect:

```
l = [4,7,9,45,-3.7,99]
arithmetic_mean(l)
```

```
---------------------------------------------------------------
---------------
NameError                                 Traceback (most r
ecent call last)
<ipython-input-29-b0693d70ca11> in <module>
      1 l = [4,7,9,45,-3.7,99]
----> 2 arithmetic_mean(l)

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

The rescue is using another asterisk:

```
arithmetic_mean(*l)
26.71666666666667
```

Class 2 exercises

Exercise 2.1 Write a program that defines a variable weight. If weight is greater than 20 (kilo's), print: "There is a $25 surcharge for luggage that is too heavy." If weight is smaller than 20, print: "Have a safe flight!" If weight is exactly 20, print: "Pfew! The weight is just right!" Make sure that you change the value of weight a couple of times to check whether your code works.

Exercise 2.2 Write code that prints all pairs (i,j) where i and j can take on the values 0 to 3, but they cannot be equal.

Exercise 2.3 Write a program that asks the user for ten numbers, and then prints the largest, the smallest, and how many are divisible by 3.

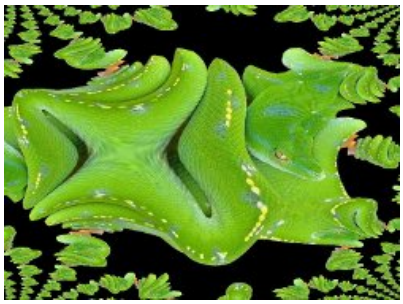
Exercise 2.4 Create a function that gets a number as parameter, and then prints the multiplication table for that number from 1 to 10. E.g., when the parameter is 12, the first line printed is "1 * 12 = 12" and the last line printed is "10 * 12 = 120."

Class 3
Lists, strings and dictionaries

SEQUENTIAL DATA TYPES

INTRODUCTION

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



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

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

```
The items or elements of strings, lists and tuples are ordered in a d
efined 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])
print(lst[2])
print(lst[-1])
```

```
Vienna
Paris
Hamburg
```

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

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

Output:

```
7
```

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

Output:

```
10
```

BYTES

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

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

Output:

```
b'Gl\xc3\xbcckliche F\xc3\xbcgung'
```

PYTHON LISTS

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

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

The main properties of Python lists:

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

LIST NOTATION AND EXAMPLES

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

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

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 differen
t!")
print("Accessing the last element of the list: " + languages[-1])
```

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

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

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

SUBLISTS

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

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

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

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

```
Marc
```

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

```
Mayer
```

```
address = person[1]
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]
```

Output:

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

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

Output:

```
'c'
```

CHANGING LIST

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

Output:

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

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

Output:

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

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

Output:

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

```
shopping_list = ['milk', 'yoghurt', 'egg', 'butter', 'bread', 'bana
nas']
```

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

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

# we go on like this:
article = shopping_list.pop()
print("shopping_list:", shopping_list)
cart.append(article)
print("cart: ", cart)
```

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

With a while loop:

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

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

```
bananas ['milk', 'yoghurt', 'egg', 'butter', 'bread'] ['ban
anas']
bread ['milk', 'yoghurt', 'egg', 'butter'] ['bananas', 'bre
ad']
butter ['milk', 'yoghurt', 'egg'] ['bananas', 'bread', 'but
ter']
egg ['milk', 'yoghurt'] ['bananas', 'bread', 'butter', 'eg
g']
yoghurt ['milk'] ['bananas', 'bread', 'butter', 'egg', 'yog
hurt']
milk [] ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'm
ilk']
shopping_list:  []
cart:  ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'mi
lk']
```

TUPLES

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

Where is the benefit of tuples?

- Tuples are faster than lists.
- If you know that some data doesn't have to be changed, you should use tuples instead of lists, because this protects your data against accidental changes.
- The main advantage of tuples is 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]
```

Output:

```
'tuples'
```

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

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

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

SLICING

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

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

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

Output:

```
'Python'
```

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

Output:

```
'n is great'
```

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

Output:

```
'Python is '
```

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

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

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

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

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

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

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

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

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

The resulting sequence consists of the following elements:

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

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

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

Output:

```
'Ph d n  e'
```

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

```
"TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCdaen
nasdeao"
```

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

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

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

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

Output:

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

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

Output:

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

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

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

Output:

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

LENGTH

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

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

Output:

```
11
```

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

Output:

```
4
```

CONCATENATION OF SEQUENCES

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

```
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
```

Output:

```
True
```

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

Output:

```
False
```

```
"e" not in abc
```

Output:

```
False
```

```
"f" not in abc
```

Output:

```
True
```

```
slogan = "Python is easy!"
"y" in slogan
```

Output:

```
True
```

```
"x" in slogan
```

Output:

```
False
```

REPETITIONS

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

str * 4

is the same as

str + str + str + str

Further examples:

```
3 * "xyz-"
```

Output:

```
'xyz-xyz-xyz-'
```

```
"xyz-" * 3
```

Output:

```
'xyz-xyz-xyz-'
```

```
3 * ["a","b","c"]
```

Output:

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

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

THE PITFALLS OF REPETITIONS

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

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

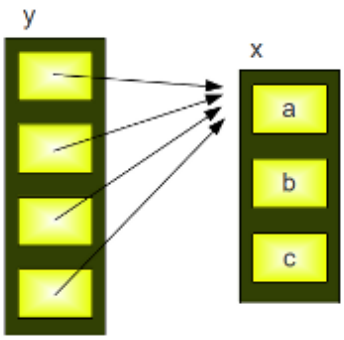
Output:

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

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

Output:

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



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

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

LISTS

CHANGING LISTS

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

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

•push
This method is used to put a new object on the stack. Depending on the point of view, we say 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 top 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]
```

Output:

```
'free'
```

POP AND APPEND

•lst.append(x)

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

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

Output:

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

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

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

```
None
```

•lst.pop(i)

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

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

Output:

```
'Hamburg'
```

```
cities
```

Output:

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

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

Output:

```
'Salzburg'
```

```
cities
```

Output:

```
['Linz', 'Vienna']
```

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

•s.pop()

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

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

Output:

```
'Strasbourg'
```

```
 cities
```

Output:

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

EXTEND

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

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

Output:

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

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

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

Output:

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

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

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

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

Now with a tuple:

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

Output:

```
['Java', 'C', 'PHP', 'C#', 'Jython', 'Python', 'IronPytho
n']
```

EXTENDING AND APPENDING LISTS WITH THE '+' OPERATOR

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

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

Output:

```
['beginner', 'intermediate', 'advanced', 'novice', 'exper
t']
```

Be careful. Never ever do the following:

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

Output:

```
[3, 4, 42]
```

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

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

Output:

```
[3, 4, 42]
```

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

Output:

```
[3, 4, 42]
```

The augmented assignment (+=) is an alternative:

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

Output:

```
[3, 4, 42]
```

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

```
import time

n= 100000

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

```
29.277812480926514
```

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

```
0.04687356948852539
```

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

```
0.03689885139465332
```

This program returns shocking results:

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

REMOVING AN ELEMENT WITH REMOVE

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

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

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

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

Output:

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

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

Output:

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

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

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

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

FIND THE POSITION OF AN ELEMENT IN A LIST

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

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

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

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

Output:

```
1
```

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

Output:

```
3
```

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

Output:

```
3
```

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

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

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

INSERT

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

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

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

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

Output:

```
['German is spoken', 'in Germany,', 'Austria', 'and', 'Swit
zerland']
```

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

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

Output:

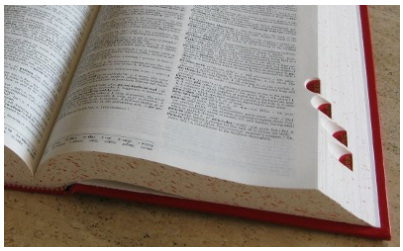
```
['a', 'b', 'c', 'd']
```

Footnotes:

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

DICTIONARIES

INTRODUCTION



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

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

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

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

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

EXAMPLES OF DICTIONARIES

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

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

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

Output:

```
8550405
```

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

Output:

```
2731571
```

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

Output:

```
667137
```

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

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

```
---------------------------------------------------------------
----------------
KeyError                                  Traceback (most r
ecent call last)
<ipython-input-4-80e422418d76> in <module>
----> 1 city_population["Detroit"]

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
```

Output:

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

Therefore it is not possible to access an element of the dictionary by a number, like we did with lists:

```
city_population[0]
```

```
---------------------------------------------------------------
----------------
KeyError                                  Traceback (most r
ecent call last)
<ipython-input-6-a4816f909f86> in <module>
----> 1 city_population[0]

KeyError: 0
```

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

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

Output:

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

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

```
city_population = {}
city_population
```

Output:

```
{}
```

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

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

Output:

```
{'bacon': 'yes', 'egg': 'yes', 'spam': 'no'}
```

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

```
food = {"bacon" : "yes", "spam" : "yes", "egg" : "yes", "spam" : "n
o" }
food
```

Output:

```
{'bacon': 'yes', 'spam': 'no', 'egg': '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"])
```

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

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

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

```
de_fr = {"rot": "rouge", "grün": "vert", "blau": "bleu", "gelb": "j
aune"}
en_de = {"red": "rot", "green": "grün", "blue": "blau", "yellow":
"gelb"}
print(en_de)
```

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

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

```
rot
```

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

```
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"}
```

```
----------------------------------------------------------------
----------------
TypeError                                 Traceback (most r
ecent call last)
<ipython-input-15-6596df6e98a2> in <module>
----> 1 dic = {[1,2,3]: "abc"}

TypeError: unhashable type: 'list'
```

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

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

Output:

```
{(1, 2, 3): 'abc', 3.1415: '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"}
de_tr = {"rot": "kırmızı", "grün": "yeşil", "blau": "mavi", "gelb":
"jel"}
en_es = {"red" : "rojo", "green" : "verde", "blue" : "azul", "yello
w":"amarillo"}

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

Output:

```
{'en_de': {'red': 'rot', 'green': 'grün', 'blue': 'blau',
'yellow': 'gelb'},
 'de_fr': {'rot': 'rouge', 'grün': 'vert', 'blau': 'bleu',
'gelb': 'jaune'},
 'de_tr': {'rot': 'kırmızı', 'grün': 'yeşil', 'blau': 'mav
i', 'gelb': 'jel'},
 'en_es': {'red': 'rojo',
  'green': 'verde',
  'blue': 'azul',
  'yellow': 'amarillo'}}
```

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

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

Output:

```
{'en_de': {'red': 'rot', 'green': 'grün', 'blue': 'blau',
'yellow': 'gelb'},
 'de_fr': {'rot': 'rouge', 'grün': 'vert', 'blau': 'bleu',
'gelb': 'jaune'},
 'de_tr': {'rot': 'kırmızı', 'grün': 'yeşil', 'blau': 'mav
i', 'gelb': 'jel'},
 'en_es': {'red': 'rojo',
  'green': 'verde',
  'blue': 'azul',
  'yellow': 'amarillo'},
 'cn_de': {'红': 'rot', '绿': 'grün', '蓝': 'blau', '黄': 'g
elb'},
 'de_ro': {'rot': 'roșu',
  'gelb': 'galben',
  'blau': 'albastru',
  'grün': 'verde'},
 'de_hex': {'rot': '#FF0000',
  'grün': '#00FF00',
  'blau': '0000FF',
  'gelb': 'FFFF00'},
 'en_pl': {'red': 'czerwony',
  'green': 'zielony',
  'blue': 'niebieski',
  'yellow': 'żółty'},
 'de_it': {'rot': 'rosso', 'gelb': 'giallo', 'blau': 'blu',
'grün': 'verde'}}
```

A dictionary of dictionaries.

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

Output:

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

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

Output:

```
{'rot': 'rouge', 'grün': 'vert', 'blau': 'bleu', 'gelb': 'j
aune'}
```

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

```
bleu
```

```
de_fr['blau']
```

Output:

```
'bleu'
```

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

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

```
red --> rot
```

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

Output:

```
'rouge'
```

```
de_fr
```

Output:

```
{'rot': 'rouge', 'grün': 'vert', 'blau': 'bleu', 'gelb': 'j
aune'}
```

```
for value in de_fr.values():
    print(value)
```

```
rouge
vert
bleu
jaune
```

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

```
rot rouge
grün vert
blau bleu
gelb jaune
```

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

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

fr_de
```

Output:

```
{'rouge': 'rot', 'vert': 'grün', 'bleu': 'blau', 'jaune':
'gelb'}
```

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

Output:

```
{'rot': '红', 'grün': '绿', 'blau': '蓝', 'gelb': '黄'}
```

OPERATORS IN DICTIONARIES

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

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

```
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" : "----.",
"." : ".-.-.-",
"," : "--..--"
}
```

```
Overwriting morsecode.py
```

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
```

Output:

```
38
```

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

```
"a" in morse
```

Output:

```
False
```

```
"A" in morse
```

Output:

```
True
```

```
"a" not in morse
```

Output:

```
True
```

```
word = input("Your word: ")

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

```
P .--.
Y -.--
T -
H ....
O ---
N -.
```

```
word = input("Your word: ")

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

print(morse_word)
```

```
.- -. -.-- -... --- -.. -.--    --- ..- -    - .... . .-. .
```

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

```
0
1
2
3
4
5
6
7
8
9
10
11
12
14
```

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].

```
en_de = {"Austria":"Vienna", "Switzerland":"Bern", "Germany":"Berli
n", "Netherlands":"Amsterdam"}
capitals = {"Austria":"Vienna", "Germany":"Berlin", "Netherlands":
"Amsterdam"}
capital = capitals.pop("Austria")
print(capital)
```

```
Vienna
```

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

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

{'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}

---------------------------------------------------------------
---------------
KeyError                                  Traceback (most r
ecent call last)
<ipython-input-43-b8864eef2b62> in <module>
      1 print(capitals)
      2 {'Netherlands': 'Amsterdam', 'Germany': 'Berlin'}
----> 3 capital = capitals.pop("Switzerland")

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)
```

Output:

```
('Toronto', 'Ontario')
```

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

```
('Olympia', 'Washington')
```

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

```
('Albany', 'New York')
```

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

```
('Lansing', 'Michigan')
```

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

```
('Boston', 'Massachusetts')
```

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"]
```

```
---------------------------------------------------------------
----------------
KeyError                                  Traceback (most r
ecent call last)
<ipython-input-52-756e77e63b38> in <module>
      1 locations = {"Toronto": "Ontario", "Vancouver": "Br
itish Columbia"}
----> 2 locations["Ottawa"]

KeyError: 'Ottawa'
```

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

```
province = "Ottawa"

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

```
Ottawa is not in locations
```

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

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

Output:

```
'Python'
```

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

Output:

```
'Perl'
```

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

```
None
```

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

Output:

```
'Python'
```

IMPORTANT METHODS

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

COPY()

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

```
{'house': 'Haus', 'cat': 'Katze'}
```

```
print(words)
```

```
{'house': 'Haus', 'cat': 'chat'}
```

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

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

trainings2 = trainings.copy()

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

```
{'course1': {'title': 'Python Training Course for Beginner
s', 'location': 'Frankfurt', 'trainer': 'Steve G. Snake'},
'course2': {'title': 'Perl Training Course for Beginners',
'location': 'Berlin', 'trainer': 'Ella M. Charming'}, 'cour
se3': {'title': 'Python Text Processing Course', 'locatio
n': 'München', 'trainer': 'Monica A. Snowdon'}}
```

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

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

```
trainings = { "course1": {"title": "Python Training Course for Begi
nners",
                          "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"])
```

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

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

CLEAR()

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 someting 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
```

Output:

```
{'Frank': {'Perl', 'Python'}, 'Monica': {'C', 'C++'}, 'Guid
o': {'Python'}}
```

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)
```

```
a
b
c
d
```

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

```
for key in d.keys():
     print(key)
```

```
a
b
c
d
```

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

```
for value in d.values():
    print(value)
```

```
123
34
304
99
```

The above loop is logically equivalent to the following one:

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

```
123
34
304
99
```

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

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

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

```
182 ns ± 1.85 ns per loop (mean ± std. dev. of 7 runs, 1000
0000 loops each)
```

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

```
131 ns ± 1.95 ns per loop (mean ± std. dev. of 7 runs, 1000
0000 loops each)
```

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": "Funk
tion"}
```

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

LISTS FROM DICTIONARIES

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

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

Output:

```
[('house', 'Haus'), ('cat', ''), ('red', 'rot')]
```

```
keys_view = w.keys()
keys = list(keys_view)
keys
```

Output:

```
['house', 'cat', 'red']
```

```
values_view = w.values()
values = list(values_view)
values
```

Output:

```
['Haus', '', 'rot']
```

```
values_view
```

Output:

```
dict_values(['Haus', '', 'rot'])
```

```
items_view
```

Output:

```
dict_items([('house', 'Haus'), ('cat', ''), ('red', 'ro
t')])
```

```
keys_view
```

Output:

```
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 the elements of the dictionary, while in the case of lists these functions have to be implemented by the programmer.

TURN LISTS INTO DICTIONARIES

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

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

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

Output:

```
<zip at 0x7f9f444073c0>
```

```
country_specialities = list(country_specialities_iterator)
print(country_specialities)
```

```
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain',
'paella'), ('USA', 'hamburger')]
```

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

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

```
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)
```

```
{'Italy': 'pizza', 'Germany': 'sauerkraut', 'Spain': 'paell
a', 'USA': 'hamburger'}
```

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))
```

Output:

```
{'Italy': 'pizza',
 'Germany': 'sauerkraut',
 'Spain': 'paella',
 'USA': 'hamburger'}
```

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)
```

```
{'Italy': 'pizza', 'Germany': 'sauerkraut', 'Spain': 'paell
a', 'USA': 'hamburger'}
```

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

EVERYTHING IN ONE STEP

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

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

```
{'pizza': 'Italy', 'sauerkraut': 'Germany', 'paella': 'Spai
n', 'hamburger': 'USA'}
```

Of course, it is more readable like this:

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

```
{'pizza': 'Italy', 'sauerkraut': 'Germany', 'paella': 'Spai
n', 'hamburger': 'USA'}
```

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)
```

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. Why do we get an empty directory by calling `dict(country_specialities_zip)` ?

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

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

```
[('pizza', 'Italy'), ('sauerkraut', 'Germany'), ('paella',
'Spain'), ('hamburger', 'USA')]
{}
```

EXERCISES

EXERCISE 1

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

EXERCISE 2

Given is the following simplified data of a supermarket:

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

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

EXERCISE 3

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

EXERCISE 4

Given is the island of Vannoth

Create a

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

EXERCISE 5

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

SOLUTIONS

SOLUTION TO EXERCISE 1

We offer two solutions to this exercise:

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

First solution:

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

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

    return merged_sum

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

dict_merge_sum(d1, d2)
```

Output:

```
{'a': 4, 'b': 5, 'd': 8}
```

Second solution:

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

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

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

dict_merge_sum(d1, d2)
```

Output:

```
{'a': 4, 'b': 5, 'd': 8}
```

SOLUTION TO EXERCISE 2:

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

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

```
The total price for buying everything:  528.37
```

SOLUTION TO EXERCISE 3:

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



customers = ["Frank", "Mary", "Paul"]
shopping_lists = {
   "Frank" : [('milk', 5), ('apples', 5), ('butter', 1), ('cookies'
, 1)],
   "Mary":  [('apples', 2), ('cheese', 4), ('bread', 2), ('pears',
3),
             ('bananas', 4), ('oranges', 1), ('cherries', 4)],
   "Paul":  [('biscuits', 2), ('apples', 3), ('yogurt', 2), ('pear
s', 1),
             ('butter', 3), ('cheese', 1), ('milk', 1), ('cookies',
4)]}


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


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

```
[('milk', 5), ('apples', 5), ('butter', 1), ('cookies', 1)]
[('apples', 2), ('cheese', 4), ('bread', 2), ('bananas',
4), ('oranges', 1)]
[('biscuits', 2), ('apples', 3), ('yogurt', 2), ('butter',
3), ('cheese', 1), ('milk', 1), ('cookies', 4)]
checkout

checkout for Frank:
  5 milk              1.19     5.95
  5 apples            3.15    15.75
  1 butter            2.29     2.29
  1 cookies           4.99     4.99
Total sum:                    28.98

checkout for Mary:
  2 apples            3.15     6.30
  4 cheese            1.90     7.60
  2 bread             2.59     5.18
  4 bananas           1.29     5.16
  1 oranges           0.99     0.99
Total sum:                    25.23

checkout for Paul:
  2 biscuits          1.45     2.90
  3 apples            3.15     9.45
  2 yogurt            3.65     7.30
  3 butter            2.29     6.87
  1 cheese            1.90     1.90
  1 milk              1.19     1.19
  4 cookies           4.99    19.96
Total sum:                    49.57
```

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

```
import random

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


articles4shopping_lists = list(supermarket.keys()) + ["pears", "che
rries"]

max_articles_per_customer = 5 # not like a real supermarket :-)
customers = ["Frank", "Mary", "Paul", "Jennifer"]
shopping_lists = {}
for customer in customers:
```

```
    no_of_items = random.randint(1, len(articles4shopping_lists))
    shopping_lists[customer] = []
    for article_name in random.sample(articles4shopping_lists, no_o
f_items):
        quantity = random.randint(1, max_articles_per_customer)
        shopping_lists[customer].append((article_name, quantity))

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

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


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

```
Shopping lists:
Frank:  [('cheese', 3), ('apples', 5)]
Mary:  [('cheese', 4), ('biscuits', 4), ('cookies', 4)]
Paul:  [('bread', 4), ('apples', 3), ('milk', 2), ('biscuit
s', 1), ('yogurt', 4), ('bananas', 4), ('cheese', 3), ('ora
nges', 2), ('cookies', 4)]
Jennifer:  [('yogurt', 1), ('oranges', 5), ('cookies', 1),
('milk', 3), ('cherries', 3), ('pears', 1), ('bananas', 1),
('biscuits', 3), ('butter', 1), ('bread', 2), ('cheese',
5), ('apples', 1)]

Checkout

checkout for Frank:
  3 cheese           1.90     5.70
  5 apples           3.15    15.75
Total sum:                   21.45

checkout for Mary:
  4 cheese           1.90     7.60
  4 biscuits         1.45     5.80
  4 cookies          4.99    19.96
Total sum:                   33.36

checkout for Paul:
  4 bread            2.59    10.36
  3 apples           3.15     9.45
  2 milk             1.19     2.38
  1 biscuits         1.45     1.45
  4 yogurt           3.65    14.60
  4 bananas          1.29     5.16
  3 cheese           1.90     5.70
  2 oranges          0.99     1.98
  4 cookies          4.99    19.96
Total sum:                   71.04

checkout for Jennifer:
  1 yogurt           3.65     3.65
  5 oranges          0.99     4.95
  1 cookies          4.99     4.99
  3 milk             1.19     3.57
  1 bananas          1.29     1.29
  3 biscuits         1.45     4.35
  1 butter           2.29     2.29
  2 bread            2.59     5.18
  5 cheese           1.90     9.50
  1 apples           3.15     3.15
Total sum:                   42.92
```

SOLUTION TO EXERCISE 4:

In [ ]:

```
rogerburgh = {'Smithstad': 5.2, 'Scottshire': 12.3, 'Clarkhaven': 1
4.9, 'Dixonshire': 12.7, 'Port Carol': 3.4}
```

SOLUTION TO EXERCISE 5:

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

FOOTNOTES

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

Class 3 exercises

Exercise 3.1 Count how many of each vowel (a, e, i, o, u) there are in a text string, and print the count for each vowel with a single formatted string. Remember that vowels can be both lower and uppercase.

Exercise 3.2 create a list consisting of the squares of the numbers 1 to 25.

Exercise 3.4 Write a program that takes a text (for instance the one given below), splits it into words (where everything that is not a letter is considered a word boundary), and case-insensitively builds a dictionary that stores for every word how often it occurs in the text. Then print all the words with their quantities in alphabetical order.

```
text = """How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood."""
```

Class 4
Read and write files

FILE MANAGEMENT

FILES IN GENERAL

It's hard to find anyone in the 21st century, who doesn't know what a file is. When we say file, we mean of course, a file on a computer. There may be people who don't know anymore the "container", like a cabinet or a folder, for keeping papers archived in a convenient order. A file on a computer is the modern counterpart of this. It is a collection of information, which can be accessed and used by a computer program. Usually, a file resides on a durable storage. Durable means that the data is persistent, i.e. it can be used by other programs after the program which has created or manipulated it, has terminated.

The term file management in the context of computers refers to the manipulation of data in a file or files and documents on a computer. Though everybody has an understanding of the term file, we present a formal definition anyway:

A file or a computer file is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

The term "file" - as we have described it in the previous paragraph - appeared in the history of computers very early. Usage can be tracked down to the year 1952, when punch cards where used.

A programming language without the capability to store and retrieve previously stored information would be hardly useful.

The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.

READING AND WRITING FILES IN PYTHON

The syntax for reading and writing files in Python is similar to programming languages like C, C++, Java, Perl, and others but a lot easier to handle.

We will start with writing a file. We have a string which contains part of the definition of a general file from Wikipedia:

```
definition = """
A computer file is a computer resource for recording data discretel
y in a
computer storage device. Just as words can be written
to paper, so can information be written to a computer
file. Files can be edited and transferred through the
internet on that particular computer system."""
```

We will write this into a file with the name `file_definition.txt` :

```
open("file_definition.txt", "w").write(definition)
```

Output:

```
283
```

If you check in your file browser, you will see a file with this name. The file will look like this: file_definition.txt

We successfully created and have written to a text file. Now, we want to see how to read this file from Python. We can read the whole text file into one string, as you can see in the following code:

```
text = open("file_definition.txt").read()
```

If you call `print(text)` , you will see the text from above again.

Reading in a text file in one string object is okay, as long as the file is not too large. If a file is large, wwe can read in the file line by line. We demonstrate how this can be achieved in the following example with a small file:

```
with open("ad_lesbiam.txt", "r") as fh:
    for line in fh:
        print(line.strip())
```

```
V. ad Lesbiam

VIVAMUS mea Lesbia, atque amemus,
rumoresque senum severiorum
omnes unius aestimemus assis!
soles occidere et redire possunt:
nobis cum semel occidit breuis lux,
nox est perpetua una dormienda.
da mi basia mille, deinde centum,
dein mille altera, dein secunda centum,
deinde usque altera mille, deinde centum.
dein, cum milia multa fecerimus,
conturbabimus illa, ne sciamus,
aut ne quis malus inuidere possit,
cum tantum sciat esse basiorum.
(GAIUS VALERIUS CATULLUS)
```

Some people don't use the with statement to read or write files. This is not a good idea. The code above without with looks like this:

```
fh = open("ad_lesbiam.txt")
for line in fh:
        print(line.strip())
fh.close()
```

A striking difference between both implementation consists in the usage of `close` . If we use `with` , we do not have to explicitly close the file. The file will be closed automatically, when the with blocks ends. Without `with` , we have to explicitly close the file, like in our second example with `fh.close()` . There is a more important difference between them: If an exception occurs inside of the `with` block, the file will be closed. If an exception occurs in the variant without `with` before the `close` , the file will not be closed. This means, you should alwawys use the `with` statement.

We saw already how to write into a file with "write". The following code is an example, in which we show how to read in from one file line by line, change the lines and write the changed content into another file. The file can be downloaded: pythonista_and_python.txt:

```
with open("pythonista_and_python.txt") as infile:
    with open("python_newbie_and_the_guru.txt", "w") as outfile:
        for line in infile:
            line = line.replace("Pythonista", "Python newbie")
            line = line.replace("Python snake", "Python guru")
            print(line.rstrip())
            # write the line into the file:
            outfile.write(line)
```

```
A blue Python newbie, green behind the ears, went to Python
ia.
She wanted to visit the famous wise green Python guru.
She wanted to ask her about the white way to avoid the blac
k.
The bright path to program in a yellow, green, or blue styl
e.
The green Python turned red, when she addressed her.
The Python newbie turned yellow in turn.
After a long but not endless loop  the wise Python uttered:
"The rainbow!"
```

As we have already mentioned: If a file is not to large and if we have to do replacements like we did in the previous example, we wouldn't read in and write out the file line by line. It is much better to use the `read` method, which returns a string containing the complete content of the file, including the carriage returns and line feeds. We can apply the changes to this string and save it into the new file. Working like this, there is no need for a `with` construct, because there will be no reference to the file, i.e. it will be immediately deleted afeter reading and writing:

```
txt = open("pythonista_and_python.txt").read()
txt = txt.replace("Pythonista", "Python newbie")
txt = txt.replace("Python snake", "Python guru")
open("python_newbie_and_the_guru.txt", "w").write(txt)
;
```

Output:

```
''
```

RESETTING THE FILES CURRENT POSITION

It's possible to set - or reset - a file's position to a certain position, also called the offset. To do this, we use the method `seek` . The parameter of seek determines the offset which we want to set the current position to. To work with `seek` , we will often need the method `tell` which "tells" us the current position. When we have just opened a file, it will be zero. Before we demonstrate the way of working of both `seek` and `tell` , we create a simple file on which we will perform our commands:

```
open("small_text.txt", "w").write("brown is her favorite colour")
;
```

Output:

```
''
```

The method `tell` returns the current stream position, i.e. the position where we will continue, when we use a "read", "readline" or so on:

```
fh = open("small_text.txt")
fh.tell()
```

Output:

```
0
```

Zero tells us that we are positioned at the first character of the file.

We will read now the next five characters of the file:

```
fh.read(5)
```

Output:

```
'brown'
```

Using `tell` again, shows that we are located at position 5:

```
fh.tell()
```

Output:

```
5
```

Using `read` without parameters will read the remainder of the file starting from this position:

```
fh.read()
```

Output:

```
' is her favorite colour'
```

Using `tell` again, tells us about the position after the last character of the file. This number corresponds to the number of characters of the file!

```
fh.tell()
```

Output:

```
28
```

With `seek` we can move the position to an arbitrary place in the file. The method `seek` takes two parameters:

```
fh.seek(offset, startpoint_for_offset)
```

where fh is the file pointer, we are working with. The parameter `offset` specifies how many positions the pointer will be moved. The question is from which position should the pointer be moved. This position is specified by the second parameter `startpoint_for_offset` . It can have the follwoing values:

```
0: reference point is the beginning of the file
1: reference point is the current file position
2: reference point is the end of the file
```

if the `startpoint_for_offset` parameter is not given, it defaults to 0.

WARNING: The values 1 and 2 for the second parameter work only, if the file has been opened for binary reading. We will cover this later!

The following examples, use the default behaviour:

```
fh.seek(13)
print(fh.tell())    # just to show you, what seek did!
fh.read()           # reading the remainder of the file
```

```
13
```

Output:

```
'favorite colour'
```

It is also possible to move the position relative to the current position. If we want to move `k` characters to the right, we can just set the argument of `seek` to `fh.tell() + k`

```
k = 6
fh.seek(5)     # setting the position to 5
fh.seek(fh.tell() + k)    #  moving k positions to the right
print("We are now at position: ", fh.tell())
```

```
We are now at position:  11
```

`seek` doesn't like negative arguments for the position. On the other hand it doesn't matter, if the value for the position is larger than the length of the file. We define a function in the following, which will set the position to zero, if a negative value is applied. As there is no efficient way to check the length of a file and because it doesn't matter, if the position is greater than the length of the file, we will keep possible values greater than the length of a file.

```
def relative_seek(fp, k):
    """ rel_seek moves the position of the file pointer k character
s to
    the left (k<0) or right (k>0)
    """
    position = fp.tell() + k
    if position < 0:
        position = 0
    fh.seek(position)


with open("small_text.txt") as fh:
    print(fh.tell())
    relative_seek(fh, 7)
    print(fh.tell())
    relative_seek(fh, -5)
    print(fh.tell())
    relative_seek(fh, -10)
    print(fh.tell())
```

```
0
7
2
0
```

You might have thought, when we wrote the function `relative_seek` why do we not use the second parameter of `seek` . After all the help file says "1 -- current stream position;". What the help file doesn't say is the fact that `seek` needs a file pointer opened with "br" (binary read), if the second parameter is set to 1 or 2. We show this in the next subchapter.

BINARY READ

So far we have only used the first parameter of `open` , i.e. the filename. The second parameter is optional and is set to "r" (read) by default. "r" means that the file is read in text mode. In text mode, if encoding (another parameter of open) is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding.

The second parameter specifies the mode of access to the file or in other words the mode in which the file is opened. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding.

We will demonstrate this in the following example. To demonstrate the different effects we need a string which uses characters which are not included in standard ASCII. This is why we use a Turkish text, because it uses many special characters and Umlaute. the English translation means "See you, I'll come tomorrow.".

We will write a file with the Turkish text "Görüşürüz, yarın geleceğim.":

```
txt = "Görüşürüz, yarın geleceğim."
number_of_chars_written = open("see_you_tomorrow.txt", "w").write(t
xt)
```

We will read in this files in text mode and binary mode to demonstrate the differences:

```
text = open("see_you_tomorrow.txt", "r").read()
print("text mode: ", text)
text_binary = open("see_you_tomorrow.txt", "rb").read()
print("binary mode: ", text_binary)
```

```
text mode:  Görüşürüz, yarın geleceğim.
binary mode:  b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbc
z, yar\xc4\xb1n gelece\xc4\x9fim.'
```

In binary mode, the characters which are not plain ASCII like "ö", "ü", "ş", "ğ" and "ı" are represented by more than one byte. In our case by two characters. 14 bytes are needed for "görüşürüz":

```
text_binary[:14]
```

Output:

```
b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'
```

"ö" for example consists of the two bytes "\xc3" and "\xb6".

```
text[:9]
```

Output:

```
'Görüşürüz'
```

There are two ways to turn a byte string into a string again:

```
t = text_binary.decode("utf-8")
print(t)
t2 = str(text_binary, "utf-8")
print(t2)
```

```
Görüşürüz, yarın geleceğim.
Görüşürüz, yarın geleceğim.
```

It is possible to use the values "1" and "2" for the second parameter of `seek` , if we open a file in binary format:

```
with open("see_you_tomorrow.txt", "rb") as fh:
    x = fh.read(14)
    print(x)
    # move 5 bytes to the right from the current position:
    fh.seek(5, 1)
    x = fh.read(3)
    print(x)
    print(str(x, "utf-8"))
    # let's move to the 8th byte from the right side of the byte st
ring:
    fh.seek(-8, 2)
    x = fh.read(5)
    print(x)
    print(str(x, "utf-8"))
```

```
b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'
b'\xc4\xb1n'
ın
b'ece\xc4\x9f'
eceğ
```

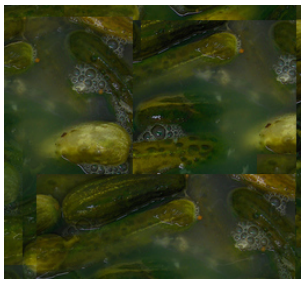
READ AND WRITE TO THE SAME FILE

In the following example we will open a file for reading and writing at the same time. If the file doesn't exist, it will be created. If you want to open an existing file for read and write, you should better use "r+", because this will not delete the content of the file.

```
fh = open('colours.txt', 'w+')
fh.write('The colour brown')

#Go to the 12th byte in the file, counting starts with 0
fh.seek(11)
print(fh.read(5))
print(fh.tell())
fh.seek(11)
fh.write('green')
fh.seek(0)
content = fh.read()
print(content)
```

```
brown
16
The colour green
```

"HOW TO GET INTO A PICKLE"



We don't really mean what the header says. On the contrary, we want to prevent any nasty situation, like losing the data, which your Python program has calculated. So, we will show you, how you can save your data in an easy way that you or better your program can reread them at a later date again. We are "pickling" the data, so that nothing gets lost.

Python offers a module for this purpose, which is called "pickle". With the algorithms of the pickle module we can serialize and de-serialize Python object structures. "Pickling" denotes the process which converts a Python object hierarchy into a byte stream, and "unpickling" on the other hand is the inverse operation, i.e. the byte stream is converted back into an object hierarchy. What we call pickling (and unpickling) is also known as "serialization" or "flattening" a data structure.

An object can be dumped with the dump method of the pickle module:

```
pickle.dump(obj, file[,protocol, *, fix_imports=True])
```

dump() writes a pickled representation of obj to the open file object file. The optional protocol argument tells the pickler to use the given protocol:

- Protocol version 0 is the original (before Python3) human-readable (ascii) protocol and is backwards compatible with previous versions of Python.
- Protocol version 1 is the old binary format which is also compatible with previous versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.
- Protocol version 3 was introduced with Python 3.0. It has explicit support for bytes and cannot be unpickled by Python 2.x pickle modules. It's the recommended protocol of Python 3.x.

The default protocol of Python3 is 3.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python3 names to the old module names used in Python2, so that the pickle data stream is readable with Python 2.

Objects which have been dumped to a file with pickle.dump can be reread into a program by using the method pickle.load(file). pickle.load recognizes automatically, which format had been used for writing the data. A simple example:

```
import pickle

cities = ["Paris", "Dijon", "Lyon", "Strasbourg"]
fh = open("data.pkl", "bw")
pickle.dump(cities, fh)
fh.close()
```

The file data.pkl can be read in again by Python in the same or another session or by a different program:

```
import pickle
f = open("data.pkl", "rb")
villes = pickle.load(f)
print(villes)
['Paris', 'Dijon', 'Lyon', 'Strasbourg']
```

Only the objects and not their names are saved. That's why we use the assignment to villes in the previous example, i.e. data = pickle.load(f).

In our previous example, we had pickled only one object, i.e. a list of French cities. But what about pickling multiple objects? The solution is easy: We pack the objects into another object, so we will only have to pickle one object again. We will pack two lists "programming_languages" and "python_dialects" into a list pickle_objects in the following example:

```
import pickle
fh = open("data.pkl","bw")
programming_languages = ["Python", "Perl", "C++", "Java", "Lisp"]
python_dialects = ["Jython", "IronPython", "CPython"]
pickle_object = (programming_languages, python_dialects)
pickle.dump(pickle_object,fh)
fh.close()
```

The pickled data from the previous example, - i.e. the data which we have written to the file data.pkl, - can be separated into two lists again, when we reread the data:

</pre> import pickle f = open("data.pkl","rb") languages, dialects) = pickle.load(f) print(languages, dialects) ['Python', 'Perl', 'C++', 'Java', 'Lisp'] ['Jython', 'IronPython', 'CPython'] </pre>

SHELVE MODULE

One drawback of the pickle module is that it is only capable of pickling one object at the time, which has to be unpickled in one go. Let's imagine this data object is a dictionary. It may be desirable that we don't have to save and load every time the whole dictionary, but save and load just a single value corresponding to just one key. The shelve module is the solution to this request. A "shelf" - as used in the shelve module - is a persistent, dictionary-like object. The difference with dbm databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the "pickle" module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys have to be strings.

The shelve module can be easily used. Actually, it is as easy as using a dictionary in Python. Before we can use a shelf object, we have to import the module. After this, we have to open a shelve object with the shelve method open. The open method opens a special shelf file for reading and writing:

</pre> import shelve s = shelve.open("MyShelve")</pre>

If the file "MyShelve" already exists, the open method will try to open it. If it isn't a shelf file, - i.e. a file which has been created with the shelve module, - we will get an error message. If the file doesn't exist, it will be created.

We can use s like an ordinary dictionary, if we use strings as keys:

```
s["street"] = "Fleet Str"
s["city"] = "London"
for key in s:
    print(key)
```

A shelf object has to be closed with the close method:

```
 s.close()
```

We can use the previously created shelf file in another program or in an interactive Python session:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informatio
n.
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 informatio
n.
    import shelve
    tele = shelve.open("MyPhoneBook")
    tele["Steve"]["phone"]
'8745'
```

EXERCISES

EXERCISE 1

Write a function which reads in a text from file and returns a list of the paragraphs. You may use one of the following books:

- Virginia Woolf: To the Lighthouse
- Samuel Butler: The Way of all Flash
- Herman Melville: Moby Dick
- David Herbert Lawrence: Sons and Lovers
- Daniel Defoe: The Life and Adventures of Robinson Crusoe
- James Joyce: Ulysses

EXERCISE 2

Save the following text containing city names and times as "cities_and_times.txt".

```
Chicago Sun 01:52
Columbus Sun 02:52
Riyadh  Sun 10:52
Copenhagen  Sun 08:52
Kuwait City Sun 10:52
Rome    Sun 08:52
Dallas  Sun 01:52
Salt Lake City  Sun 01:52
San Francisco Sun 00:52
Amsterdam Sun 08:52
Denver Sun 01:52
San Salvador Sun 01:52
Detroit Sun 02:52
Las Vegas Sun 00:52
Santiago    Sun 04:52
Anchorage Sat 23:52
Ankara Sun 10:52
Lisbon  Sun 07:52
São Paulo   Sun 05:52
Dubai   Sun 11:52
London  Sun 07:52
Seattle Sun 00:52
Dublin  Sun 07:52
Los Angeles Sun 00:52
Athens  Sun 09:52
Edmonton Sun 01:52
Madrid  Sun 08:52
Shanghai Sun 15:52
Atlanta Sun 02:52
Frankfurt   Sun 08:52
Singapore Sun 15:52
Auckland Sun 20:52
Halifax Sun 03:52
Melbourne Sun 18:52
Stockholm   Sun 08:52
Barcelona   Sun 08:52
Miami   Sun 02:52
Minneapolis Sun 01:52
Sydney Sun 18:52
Beirut  Sun 09:52
Helsinki    Sun 09:52
Montreal    Sun 02:52
```

```
Berlin  Sun 08:52
Houston Sun 01:52
Moscow  Sun 10:52
Indianapolis    Sun 02:52
Boston  Sun 02:52
Tokyo   Sun 16:52
Brasilia Sun 05:52
Istanbul Sun 10:52
Toronto Sun 02:52
Vancouver   Sun 00:52
Brussels    Sun 08:52
Jerusalem   Sun 09:52
New Orleans Sun 01:52
Vienna  Sun 08:52
Bucharest   Sun 09:52
Johannesburg    Sun 09:52
New York    Sun 02:52
Warsaw  Sun 08:52
Budapest    Sun 08:52
Oslo    Sun 08:52
Washington DC   Sun 02:52
Ottawa  Sun 02:52
Winnipeg    Sun 01:52
Cairo   Sun 09:52
Paris   Sun 08:52
Calgary Sun 01:52
Kathmandu   Sun 13:37
Philadelphia    Sun 02:52
Zurich  Sun 08:52
Cape Town   Sun 09:52
Phoenix Sun 00:52
Prague  Sun 08:52
Casablanca  Sun 07:52
Reykjavik   Sun 07:52
```

Each line contains the name of the city, followed by the name of the day ("Sun") and the time in the form hh:mm. Read in the file and create an alphabetically ordered list of the form

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)), ('Ank
ara', 'Sun', (10, 52)), ('Athens', 'Sun', (9, 52)), ('Atlanta', 'Su
n', (2, 52)), ('Auckland', 'Sun', (20, 52)), ('Barcelona', 'Sun', (8,
52)), ('Beirut', 'Sun', (9, 52)),

...

 ('Toronto', 'Sun', (2, 52)), ('Vancouver', 'Sun', (0, 52)), ('Vienn
a', 'Sun', (8, 52)), ('Warsaw', 'Sun', (8, 52)), ('Washington DC', 'S
un', (2, 52)), ('Winnipeg', 'Sun', (1, 52)), ('Zurich', 'Sun', (8, 5
2))]
```

Finally, the list should be dumped for later usage with the pickle module. We will use this list in our chapter on Numpy dtype.

SOLUTIONS

SOLUTION 1

```
def text2paragraphs(filename, min_size=1):
    """ A text contained in the file 'filename' will be read
    and chopped into paragraphs.
    Paragraphs with a string length less than min_size will be igno
red.
    A list of paragraph strings will be returned"""

    txt = open(filename).read()
    paragraphs = [para for para in txt.split("\n\n") if len(para) >
min_size]
    return paragraphs

paragraphs = text2paragraphs("books/to_the_lighthouse_woolf.txt", m
in_size=100)

for i in range(10, 14):
    print(paragraphs[i])
```

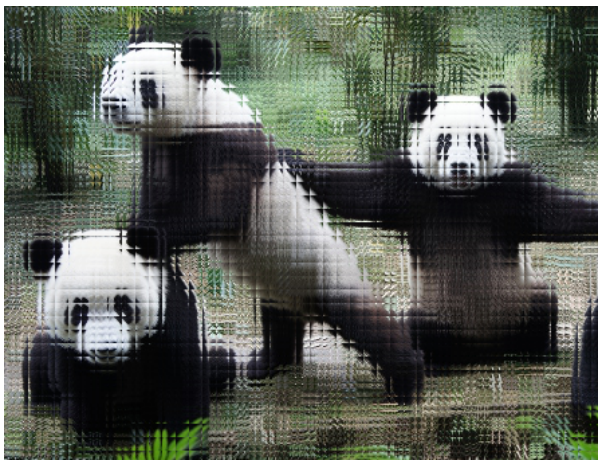
“I should think there would be no one to talk to in Manches
ter,” she
replied at random. Mr. Fortescue had been observing her for
a moment or
two, as novelists are inclined to observe, and at this rema
rk he smiled,
and made it the text for a little further speculation.
“In spite of a slight tendency to exaggeration, Katharine d
ecidedly
hits the mark,” he said, and lying back in his chair, with
his opaque
contemplative eyes fixed on the ceiling, and the tips of hi
s fingers
pressed together, he depicted, first the horrors of the str
eets of
Manchester, and then the bare, immense moors on the outskir
ts of the
town, and then the scrubby little house in which the girl w
ould live,
and then the professors and the miserable young students de
voted to the
more strenuous works of our younger dramatists, who would v
isit her,
and how her appearance would change by degrees, and how she
would fly to
London, and how Katharine would have to lead her about, as
one leads an
eager dog on a chain, past rows of clamorous butchers’ shop
s, poor dear
creature.
“Oh, Mr. Fortescue,” exclaimed Mrs. Hilbery, as he finishe
d, “I had just
written to say how I envied her! I was thinking of the big
gardens and
the dear old ladies in mittens, who read nothing but the “S
pectator,”
and snuff the candles. Have they ALL disappeared? I told h
er she would
find the nice things of London without the horrid streets t
hat depress
one so.”
“There is the University,” said the thin gentleman, who had
previously
insisted upon the existence of people knowing Persian.

SOLUTION 2

```
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".

Introduction into Pandas



The pandas we are writing about in this chapter have nothing to do with the cute panda bears. Endearing bears are not what our visitors expect in a Python tutorial. Pandas is the name for a Python module, which is rounding up the capabilities of Numpy, Scipy and Matplotlab. The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".

There is often some confusion about whether Pandas is an alternative to Numpy, SciPy and Matplotlib. The truth is that it is built on top of Numpy. This means that Numpy is required by pandas. Scipy and Matplotlib on the other hand are not required by pandas but they are extremely useful. That's why the Pandas project lists them as "optional dependency".

Pandas is a software library written for the Python programming language. It is used for data manipulation and analysis. It provides special data structures and operations for the manipulation of numerical tables and time series. Pandas is free software released under the three-clause BSD license.

DATA STRUCTURES

We will start with the following two important data structures of Pandas:

- Series and
- DataFrame

SERIES

A Series is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

We define a simple Series object in the following example by instantiating a Pandas Series object with a list. We will later see that we can use other data objects for example Numpy arrays and dictionaries as well to instantiate a Series object.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
S
```

Output: :

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

We haven't defined an index in our example, but we see two columns in our output: The right column contains our data, whereas the left column contains the index. Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

We can directly access the index and the values of our Series S:

```
print(S.index)
print(S.values)
```

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

If we compare this to creating an array in numpy, we will find lots of similarities:

```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

So far our Series have not been very different to ndarrays of Numpy. This changes, as soon as we start defining Series objects with individual indices:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
S
```

Output: :

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

A big advantage to NumPy arrays is obvious from the previous example: We can use arbitrary indices.

If we add two series with the same indices, we get a new series with the same index and the correponding values will be added:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

In principle, the indices can be completely different, as in the following example. We have two indices. One is the Turkish translation of the English fruit names:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

```
apples     NaN
armut      NaN
cherries   NaN
elma       NaN
kiraz      NaN
oranges    NaN
pears      NaN
portakal   NaN
dtype: float64
```

INDEXING

It's possible to access single values of a Series.

```
print(S['apples'])
```

```
20
```

This lookes like accessing the values of dictionaries through keys.

However, Series objects can also be accessed by multiple indexes at the same time. This can be done by packing the indexes into a list. This type of access returns a Pandas Series again:

```
print(S[['apples', 'oranges', 'cherries']])
```

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Similar to Numpy we can use scalar operations or mathematical functions on a series:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

```
apples       92
oranges     144
cherries    220
pears        52
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

PANDAS.SERIES.APPLY

Series.apply(func, convert_dtype=True, args=(), **kwds)

The function "func" will be applied to the Series and it returns either a Series or a DataFrame, depending on "func".

Parameter	Meaning
func	a function, which can be a NumPy function that will be applied to the entire Series or a Python function that will be applied to every single value of the series
convert_dtype	A boolean value. If it is set to True (default), apply will try to find better dtype for elementwise function results. If False, leave as dtype=object
args	Positional arguments which will be passed to the function "func" additionally to the values from the series.
**kwds	Additional keyword arguments will be passed as keywords to the function

Example:

```
S.apply(np.log)
```

Output: :

```
apples      2.995732
oranges     3.496508
cherries    3.951244
pears       2.302585
dtype: float64
```

We can also use Python lambda functions. Let's assume, we have the following task. The test the amount of fruit for every kind. If there are less than 50 available, we will augment the stock by 10:

```
S.apply(lambda x: x if x > 50 else x+10 )
```

Output: :

```
apples      30
oranges     43
cherries    52
pears       20
dtype: int64
```

FILTERING WITH A BOOLEAN ARRAY

Similar to numpy arrays, we can filter Pandas Series with a Boolean array:

```
S[S>30]
```

Output: :

```
oranges     33
cherries    52
dtype: int64
```

A series can be seen as an ordered Python dictionary with a fixed length.

```
"apples" in S
```

Output: :

```
True
```

CREATING SERIES OBJECTS FROM DICTIONARIES

We can even use a dictionary to create a Series object. The resulting Series contains the dict's keys as the indices and the values as the values.

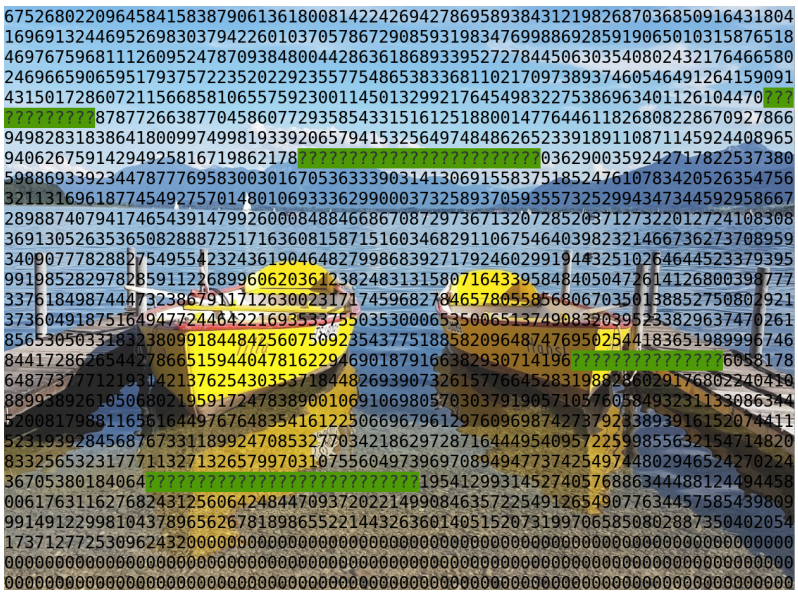
```
cities = {"London":     8615246,
          "Berlin":     3562166,
          "Madrid":     3165235,
          "Rome":       2874038,
          "Paris":      2273305,
          "Vienna":     1805681,
          "Bucharest":  1803425,
          "Hamburg":    1760433,
          "Budapest":   1754000,
          "Warsaw":     1740119,
          "Barcelona":  1602386,
          "Munich":     1493900,
          "Milan":      1350680}

city_series = pd.Series(cities)
print(city_series)
```

```
London        8615246
Berlin        3562166
Madrid        3165235
Rome          2874038
Paris         2273305
Vienna        1805681
Bucharest     1803425
Hamburg       1760433
Budapest      1754000
Warsaw        1740119
Barcelona     1602386
Munich        1493900
Milan         1350680
dtype: int64
```

NAN - MISSING DATA

One problem in dealing with data analysis tasks consists in missing data. Pandas makes it as easy as possible to work with missing data.

If we look once more at our previous example, we can see that the index of our series is the same as the keys of the dictionary we used to create the cities_series. Now, we want to use an index which is not overlapping with the dictionary keys. We have already seen that we can pass a list or a tuple to the keyword argument 'index' to define the index. In our next example, the list (or tuple) passed to the keyword parameter 'index' will not be equal to the keys. This means that some cities from the dictionary will be missing and two cities ("Zurich" and "Stuttgart") don't occur in the dictionary.



```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Output: :

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

Due to the Nan values the population values for the other cities are turned into floats. There is no missing data in the following examples, so the values are int:

```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Output: :

```
London     8615246
Paris      2273305
Berlin     3562166
Hamburg    1760433
dtype: int64
```

THE METHODS ISNULL() AND NOTNULL()

We can see, that the cities, which are not included in the dictionary, get the value NaN assigned. NaN stands for "not a number". It can also be seen as meaning "missing" in our example.

We can check for missing values with the methods isnull and notnull:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
print(my_city_series.isnull())
```

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

```
print(my_city_series.notnull())
```

```
London         True
Paris          True
Zurich        False
Berlin         True
Stuttgart     False
Hamburg        True
dtype: bool
```

CONNECTION BETWEEN NAN AND NONE

We get also a NaN, if a value in the dictionary has a None:

```
d = {"a":23, "b":45, "c":None, "d":0}
S = pd.Series(d)
print(S)
```

```
a    23.0
b    45.0
c     NaN
d     0.0
dtype: float64
```

```
pd.isnull(S)
```

Output: :

```
a    False
b    False
c     True
d    False
dtype: bool
```

```
pd.notnull(S)
```

Output: :

```
a     True
b     True
c    False
d     True
dtype: bool
```

FILTERING OUT MISSING DATA

It's possible to filter out missing data with the Series method dropna. It returns a Series which consists only of non-null data:

```
print(my_city_series.dropna())
```

```
London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

FILLING IN MISSING DATA

In many cases you don't want to filter out missing data, but you want to fill in appropriate data for the empty gaps. A suitable method in many situations will be fillna:

```
print(my_city_series.fillna(0))
```

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

Okay, that's not what we call "fill in appropriate data for the empty gaps". If we call fillna with a dict, we can provide the appropriate data, i.e. the population of Zurich and Stuttgart:

```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities)
```

Output: :

```
London       8615246.0
Paris        2273305.0
Zurich        378884.0
Berlin       3562166.0
Stuttgart     597939.0
Hamburg      1760433.0
dtype: float64
```

We still have the problem that integer values - which means values which should be integers like number of people - are converted to float as soon as we have NaN values. We can solve this problem now with the method 'fillna':

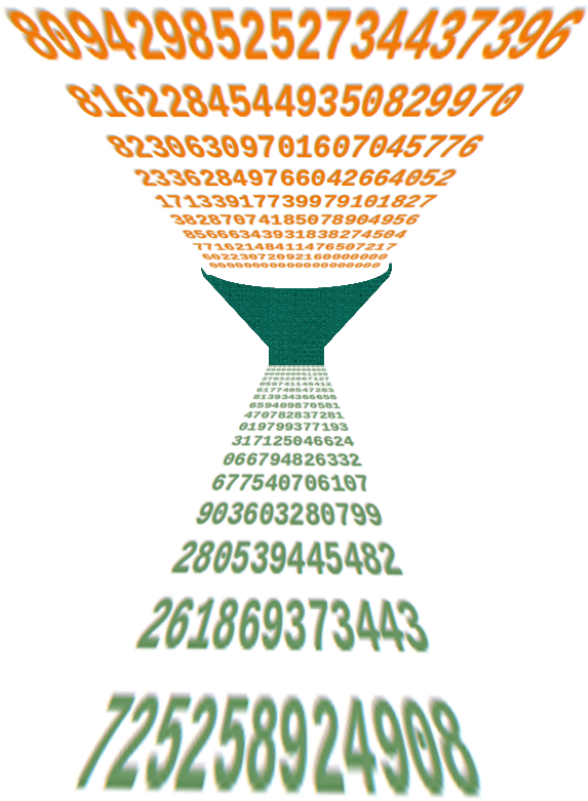
```
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
          "Bucharest":1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona":1602386,
          "Munich":    1493900,
          "Milan":     1350680}

my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series = my_city_series.fillna(0).astype(int)
print(my_city_series)
```

```
London       8615246
Paris        2273305
Zurich             0
Berlin       3562166
Stuttgart          0
Hamburg      1760433
dtype: int64
```

READING AND WRITING DATA

All the powerful data structures like the Series and the DataFrames would avail to nothing, if the Pandas module wouldn't provide powerful functionalities for reading in and writing out data. It is not only a matter of having a functions for interacting with files. To be useful to data scientists it also needs functions which support the most important data formats like

- Delimiter-separated files, like e.g. csv
- Microsoft Excel files
- HTML
- XML
- JSON



DELIMITER-SEPARATED VALUES

Most people take csv files as a synonym for delimter-separated values files. They leave the fact out of account that csv is an acronym for "comma separated values", which is not the case in many situations. Pandas also uses "csv" and contexts, in which "dsv" would be more appropriate.

Delimiter-separated values (DSV) are defined and stored two-dimensional arrays (for example strings) of data by separating the values in each row with delimiter characters defined for this purpose. This way of implementing data is often used in combination of spreadsheet programs, which can read in and write out data as DSV. They are also used as a general data exchange format.

We call a text file a "delimited text file" if it contains text in DSV format.

For example, the file dollar_euro.txt is a delimited text file and uses tabs (\t) as delimiters.

READING CSV AND DSV FILES

Pandas offers two ways to read in CSV or DSV files to be precise:

- DataFrame.from_csv
- read_csv

There is no big difference between those two functions, e.g. they have different default values in some cases and read_csv has more paramters. We will focus on read_csv, because DataFrame.from_csv is kept inside Pandas for reasons of backwards compatibility.

```
import pandas as pd

exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t")
print(exchange_rates)
```

```
    Year   Average  Min USD/EUR  Max USD/EUR  Working days
0   2016  0.901696     0.864379     0.959785           247
1   2015  0.901896     0.830358     0.947688           256
2   2014  0.753941     0.716692     0.823655           255
3   2013  0.753234     0.723903     0.783208           255
4   2012  0.778848     0.743273     0.827198           256
5   2011  0.719219     0.671953     0.775855           257
6   2010  0.755883     0.686672     0.837381           258
7   2009  0.718968     0.661376     0.796495           256
8   2008  0.683499     0.625391     0.802568           256
9   2007  0.730754     0.672314     0.775615           255
10  2006  0.797153     0.750131     0.845594           255
11  2005  0.805097     0.740357     0.857118           257
12  2004  0.804828     0.733514     0.847314           259
13  2003  0.885766     0.791766     0.963670           255
14  2002  1.060945     0.953562     1.165773           255
15  2001  1.117587     1.047669     1.192748           255
16  2000  1.085899     0.962649     1.211827           255
17  1999  0.939475     0.848176     0.998502           261
```

As we can see, read_csv used automatically the first line as the names for the columns. It is possible to give other names to the columns. For this purpose, we have to skip the first line by setting the parameter "header" to 0 and we have to assign a list with the column names to the parameter "names":

```
import pandas as pd

exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t",
                             header=0,
                             names=["year", "min", "max", "days"])
print(exchange_rates)
```

```
          year       min       max  days
2016  0.901696  0.864379  0.959785   247
2015  0.901896  0.830358  0.947688   256
2014  0.753941  0.716692  0.823655   255
2013  0.753234  0.723903  0.783208   255
2012  0.778848  0.743273  0.827198   256
2011  0.719219  0.671953  0.775855   257
2010  0.755883  0.686672  0.837381   258
2009  0.718968  0.661376  0.796495   256
2008  0.683499  0.625391  0.802568   256
2007  0.730754  0.672314  0.775615   255
2006  0.797153  0.750131  0.845594   255
2005  0.805097  0.740357  0.857118   257
2004  0.804828  0.733514  0.847314   259
2003  0.885766  0.791766  0.963670   255
2002  1.060945  0.953562  1.165773   255
2001  1.117587  1.047669  1.192748   255
2000  1.085899  0.962649  1.211827   255
1999  0.939475  0.848176  0.998502   261
```

EXERCISE

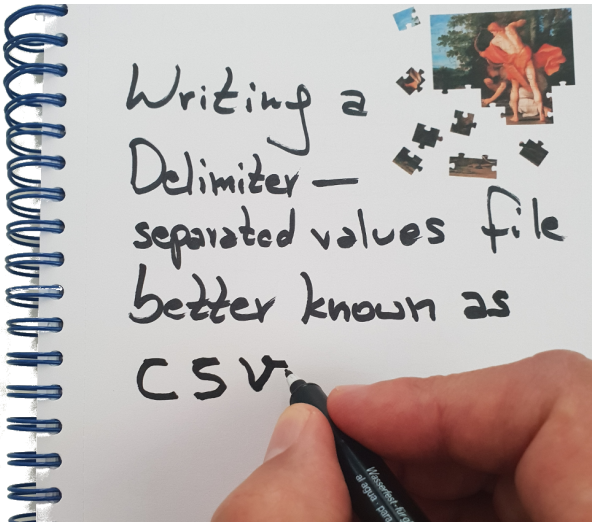
The file "countries_population.csv" is a csv file, containing the population numbers of all countries (July 2014). The delimiter of the file is a space and commas are used to separate groups of thousands in the numbers. The method 'head(n)' of a DataFrame can be used to give out only the first n rows or lines. Read the file into a DataFrame.

Solution:

```
pop = pd.read_csv("data1/countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")
print(pop.head(5))
```

```
                Population
Country
China           1355692576
India           1236344631
European Union   511434812
United States    318892103
Indonesia        253609643
```

WRITING CSV FILES



We can create csv (or dsv) files with the method "to_csv". Before we do this, we will prepare some data to output, which we will write to a file. We have two csv files with population data for various countries. countries_male_population.csv contains the figures of the male populations and countries_female_population.csv correspondingly the numbers for the female populations. We will create a new csv file with the sum:

```
column_names = ["Country"] + list(range(2002, 2013))
male_pop = pd.read_csv("data1/countries_male_population.csv",
                       header=None,
                       index_col=0,
                       names=column_names)

female_pop = pd.read_csv("data1/countries_female_population.csv",
                         header=None,
                         index_col=0,
                         names=column_names)


population = male_pop + female_pop
```

```
population
```

Output::

Country	2002	2003	2004	2005	2006	2007	2008	2
Australia	19640979.0	19872646	20091504	20339759	20605488	21015042	21431781	21874
Austria	8139310.0	8067289	8140122	8206524	8265925	8298923	8331930	8355
Belgium	10309725.0	10355844	10396421	10445852	10511382	10584534	10666866	10753
Canada	NaN	31361611	31372587	31989454	32299496	32649482	32927372	33327
Czech Republic	10269726.0	10203269	10211455	10220577	10251079	10287189	10381130	10467
Denmark	5368354.0	5383507	5397640	5411405	5427459	5447084	5475791	5511
Finland	5194901.0	5206295	5219732	5236611	5255580	5276955	5300484	5326
France	59337731.0	59630121	59900680	62518571	62998773	63392140	63753140	64366
Germany	82440309.0	82536680	82531671	82500849	82437995	82314906	82217837	82002
Greece	10988000.0	11006377	11040650	11082751	11125179	11171740	11213785	11260
Hungary	10174853.0	10142362	10116742	10097549	10076581	10066158	10045401	10030
Iceland	286575.0	288471	290570	293577	299891	307672	315459	319
Ireland	3882683.0	3963636	4027732	4109173	4209019	4239848	4401335	4450
Italy	56993742.0	57321070	57888245	58462375	58751711	59131287	59619290	60045
Japan	127291000.0	127435000	127620000	127687000	127767994	127770000	127771000	12769
Korea	47639618.0	47925318	48082163	48138077	48297184	48456369	48606787	48746
Luxembourg	444050.0	448300	451600	455000	469086	476187	483799	493
Mexico	101826249.0	103039964	104213503	103001871	103946866	104874282	105790725	10668
Netherlands	16105285.0	16192572	16258032	16305526	16334210	16357992	16405399	16485
New Zealand	3939130.0	4009200	4062500	4100570	4139470	4228280	4268880	4315
Norway	4524066.0	4552252	4577457	4606363	4640219	4681134	4737171	4799
Poland	38632453.0	38218531	38190608	38173835	38157055	38125479	38115641	38135
Portugal	10335559.0	10407465	10474685	10529255	10569592	10599095	10617575	10627
Slovak Republic	5378951.0	5379161	5380053	5384822	5389180	5393637	5400998	5412
Spain	40409330.0	41550584	42345342	43038035	43758250	44474631	45283259	45828
Sweden	8909128.0	8940788	8975670	9011392	9047752	9113257	9182927	9256
Switzerland	7261210.0	7313853	7364148	7415102	7459128	7508739	7593494	7701
Turkey	NaN	70171979	70689500	71607500	72519974	72519974	70586256	71517
United Kingdom	58706905.0	59262057	59699828	60059858	60412870	60781346	61179260	61595
United States	277244916.0	288774226	290810719	294442683	297308143	300184434	304846731	30512

```
population.to_csv("data1/countries_total_population.csv")
```

We want to create a new DataFrame with all the information, i.e. female, male and complete population. This means that we have to introduce an hierarchical index. Before we do it on our DataFrame, we will introduce this problem in a simple example:

```
import pandas as pd

shop1 = {"foo":{2010:23, 2011:25}, "bar":{2010:13, 2011:29}}
shop2 = {"foo":{2010:223, 2011:225}, "bar":{2010:213, 2011:229}}

shop1 = pd.DataFrame(shop1)
shop2 = pd.DataFrame(shop2)
both_shops = shop1 + shop2
print("Sales of shop1:\n", shop1)
print("\nSales of both shops\n", both_shops)
```

```
Sales of shop1:
       foo  bar
2010   23   13
2011   25   29

Sales of both shops
       foo  bar
2010  246  226
2011  250  258
```

```
shops = pd.concat([shop1, shop2], keys=["one", "two"])
shops
```

Output::

		foo	bar
one	**2010**	23	13
	2011	25	29
two	**2010**	223	213
	2011	225	229

We want to swap the hierarchical indices. For this we will use 'swaplevel':

```
shops.swaplevel()
shops.sort_index(inplace=True)
shops
```

Output::

		foo	bar
one	**2010**	23	13
	2011	25	29
two	**2010**	223	213
	2011	225	229

We will go back to our initial problem with the population figures. We will apply the same steps to those DataFrames:

```
pop_complete = pd.concat([population.T,
                          male_pop.T,
                          female_pop.T],
                          keys=["total", "male", "female"])
df = pop_complete.swaplevel()
df.sort_index(inplace=True)
df[["Austria", "Australia", "France"]]
```

Output::

	Country	Austria	Australia	France
2002	**female**	4179743.0	9887846.0	30510073.0
	male	3959567.0	9753133.0	28827658.0
	total	8139310.0	19640979.0	59337731.0
2003	**female**	4158169.0	9999199.0	30655533.0
	male	3909120.0	9873447.0	28974588.0
	total	8067289.0	19872646.0	59630121.0
2004	**female**	4190297.0	10100991.0	30789154.0
	male	3949825.0	9990513.0	29111526.0
	total	8140122.0	20091504.0	59900680.0
2005	**female**	4220228.0	10218321.0	32147490.0
	male	3986296.0	10121438.0	30371081.0
	total	8206524.0	20339759.0	62518571.0
2006	**female**	4246571.0	10348070.0	32390087.0
	male	4019354.0	10257418.0	30608686.0
	total	8265925.0	20605488.0	62998773.0
2007	**female**	4261752.0	10570420.0	32587979.0
	male	4037171.0	10444622.0	30804161.0
	total	8298923.0	21015042.0	63392140.0
2008	**female**	4277716.0	10770864.0	32770860.0
	male	4054214.0	10660917.0	30982280.0
	total	8331930.0	21431781.0	63753140.0
2009	**female**	4287213.0	10986535.0	33208315.0
	male	4068047.0	10888385.0	31158647.0
	total	8355260.0	21874920.0	64366962.0
2010	**female**	4296197.0	11218144.0	33384930.0
	male	4079093.0	11124254.0	31331380.0
	total	8375290.0	22342398.0	64716310.0
2011	**female**	4308915.0	11359807.0	33598633.0
	male	4095337.0	11260747.0	31531113.0
	total	8404252.0	22620554.0	65129746.0
2012	**female**	4324983.0	11402769.0	33723892.0
	male	4118035.0	11280804.0	31670391.0
	total	8443018.0	22683573.0	65394283.0

```
df.to_csv("data1/countries_total_population.csv")
```

EXERCISE

- Read in the dsv file (csv) bundeslaender.txt. Create a new file with the columns 'land', 'area', 'female', 'male', 'population' and 'density' (inhabitants per square kilometres.
- print out the rows where the area is greater than 30000 and the population is greater than 10000
- Print the rows where the density is greater than 300

```
lands = pd.read_csv('data1/bundeslaender.txt', sep=" ")
print(lands.columns.values)
```

```
['land' 'area' 'male' 'female']
```

```
# swap the columns of our DataFrame:
lands = lands.reindex(columns=['land', 'area', 'female', 'male'])
lands[:2]
```

Output::

	land	area	female	male
0	Baden-Württemberg	35751.65	5465	5271
1	Bayern	70551.57	6366	6103

```
lands.insert(loc=len(lands.columns),
             column='population',
             value=lands['female'] + lands['male'])

lands[:3]
```

Output::

	land	area	female	male	population
0	Baden-Württemberg	35751.65	5465	5271	10736
1	Bayern	70551.57	6366	6103	12469
2	Berlin	891.85	1736	1660	3396

```
lands.insert(loc=len(lands.columns),
             column='density',
             value=(lands['population'] * 1000 / lands['area']).rou
nd(0))

lands[:4]
```

Output::

	land	area	female	male	population	density
0	Baden-Württemberg	35751.65	5465	5271	10736	300.0
1	Bayern	70551.57	6366	6103	12469	177.0
2	Berlin	891.85	1736	1660	3396	3808.0
3	Brandenburg	29478.61	1293	1267	2560	87.0

```
print(lands.loc[(lands.area>30000) & (lands.population>10000)])
```

```
                  land      area  female  male  population
density
0    Baden-Württemberg  35751.65    5465  5271       10736
300.0
1               Bayern  70551.57    6366  6103       12469
177.0
9  Nordrhein-Westfalen  34085.29    9261  8797       18058
530.0
```

READING AND WRITING EXCEL FILES

It is also possible to read and write Microsoft Excel files. The Pandas functionalities to read and write Excel files use the modules 'xlrd' and 'openpyxl'. These modules are not automatically installed by Pandas, so you may have to install them manually!

We will use a simple Excel document to demonstrate the reading capabilities of Pandas. The document sales.xls contains two sheets, one called 'week1' and the other one 'week2'.
An Excel file can be read in with the Pandas function "read_excel". This is demonstrated in the following example Python code:

```
excel_file = pd.ExcelFile("data1/sales.xls")

sheet = pd.read_excel(excel_file)
sheet
```

Output::

	Weekday	Sales
0	Monday	123432.980000
1	Tuesday	122198.650200
2	Wednesday	134418.515220
3	Thursday	131730.144916
4	Friday	128173.431003

The document "sales.xls" contains two sheets, but we only have been able to read in the first one with "read_excel". A complete Excel document, which can consist of an arbitrary number of sheets, can be completely read in like this:

```
docu = {}
for sheet_name in excel_file.sheet_names:
    docu[sheet_name] = excel_file.parse(sheet_name)

for sheet_name in docu:
    print("\n" + sheet_name + ":\n", docu[sheet_name])
```

```
week1:
       Weekday           Sales
0       Monday  123432.980000
1      Tuesday  122198.650200
2    Wednesday  134418.515220
3     Thursday  131730.144916
4       Friday  128173.431003

week2:
       Weekday           Sales
0       Monday  223277.980000
1      Tuesday  234441.879000
2    Wednesday  246163.972950
3     Thursday  241240.693491
4       Friday  230143.621590
```

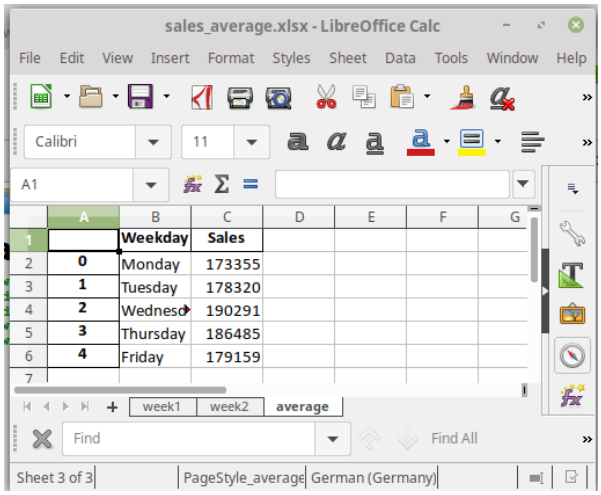
We will calculate now the avarage sales numbers of the two weeks:

```
average = docu["week1"].copy()
average["Sales"] = (docu["week1"]["Sales"] + docu["week2"]["Sales"
]) / 2
print(average)
```

```
       Weekday           Sales
0       Monday  173355.480000
1      Tuesday  178320.264600
2    Wednesday  190291.244085
3     Thursday  186485.419203
4       Friday  179158.526297
```

We will save the DataFrame 'average' in a new document with 'week1' and 'week2' as additional sheets as well:

```
writer = pd.ExcelWriter('data1/sales_average.xlsx')
document['week1'].to_excel(writer,'week1')
document['week2'].to_excel(writer,'week2')
average.to_excel(writer,'average')
writer.save()
writer.close()
```


sales_average.xlsx - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Calibri
11
A1
Weekday
Sales
0 Monday 173355
1 Tuesday 178320
2 Wednesd 190291
3 Thursday 186485
4 Friday 179159
week1 week2 average
Find
Find All
Sheet 3 of 3
PageStyle_average
German (Germany)

Class 4 exercises

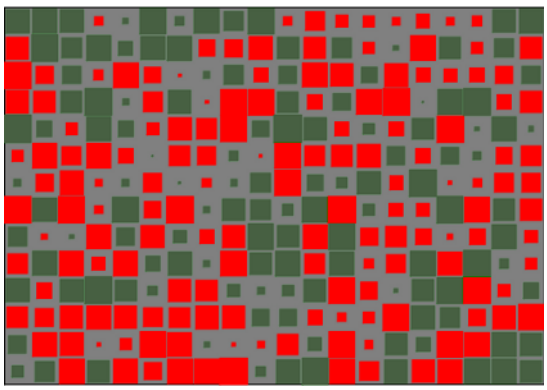
Exercise 4.1 Modify the program of the exercise 3.3 to read the text from a file and then it writes the results in a file named occurences.txt.

Exercise 4.2 Load the contents from the file “pc_inventory.csv” using Pandas and put them in a list of lists (each line in the file being one list in the list of lists).

Class 5
Numerical programming

NUMPY TUTORIAL

INTRODUCTION



NumPy is a module for Python. The name is an acronym for "Numeric Python" or "Numerical Python". It is pronounced /ˈnʌmpaɪ/ (NUM-py) or less often /ˈnʌmpi (NUM-pee)). It is an extension module for Python, mostly written in C. This makes sure that the precompiled mathematical and numerical functions and functionalities of Numpy guarantee great execution speed.

Furthermore, NumPy enriches the programming language Python with powerful data structures, implementing multi-dimensional arrays and matrices. These data structures guarantee efficient calculations with matrices and arrays. The implementation is even aiming at huge matrices and arrays, better know under the heading of "big data". Besides that the module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.

SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy needs Numpy, as it is based on the data structures of Numpy and furthermore its basic creation and manipulation functions. It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.

Both NumPy and SciPy are not part of a basic Python installation. They have to be installed after the Python installation. NumPy has to be installed before installing SciPy.

(Comment: The diagram of the image on the right side is the graphical visualisation of a matrix with 14 rows and 20 columns. It's a so-called Hinton diagram. The size of a square within this diagram corresponds to the size of the value of the depicted matrix. The colour determines, if the value is positive or negative. In our example: the colour red denotes negative values and the colour green denotes positive values.)

NumPy is based on two earlier Python modules dealing with arrays. One of these is Numeric. Numeric is like NumPy a Python module for high-performance, numeric computing, but it is obsolete nowadays. Another predecessor of NumPy is Numarray, which is a complete rewrite of Numeric but is deprecated as well. NumPy is a merger of those two, i.e. it is build on the code of Numeric and the features of Numarray.

COMPARISON BETWEEN CORE PYTHON AND NUMPY

When we say "Core Python", we mean Python without any special modules, i.e. especially without NumPy.

The advantages of Core Python:

- high-level number objects: integers, floating point
- containers: lists with cheap insertion and append methods, dictionaries with fast lookup

Advantages of using Numpy with Python:

- array oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation

A SIMPLE NUMPY EXAMPLE

Before we can use NumPy we will have to import it. It has to be imported like any other module:

```
import numpy
```

But you will hardly ever see this. Numpy is usually renamed to np:

```
import numpy as np
```

Our first simple Numpy example deals with temperatures. Given is a list with values, e.g. temperatures in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 2
0.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

```
C = np.array(cvalues)
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
print(C * 9 / 5 + 32)
```

```
[68.18 69.44 71.42 72.5  72.86 72.14 71.24 70.16 69.62 68.1
8]
```

The array C has not been changed by this expression:

```
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Compared to this, the solution for our Python list looks awkward:

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

```
[68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24000000000000
1, 70.16, 69.62, 68.18]
```

So far, we referred to C as an array. The internal type is "ndarray" or to be even more precise "C is an instance of the class numpy.ndarray":

```
type(C)
```

Output: :

```
numpy.ndarray
```

In the following, we will use the terms "array" and "ndarray" in most cases synonymously.

GRAPHICAL REPRESENTATION OF THE VALUES

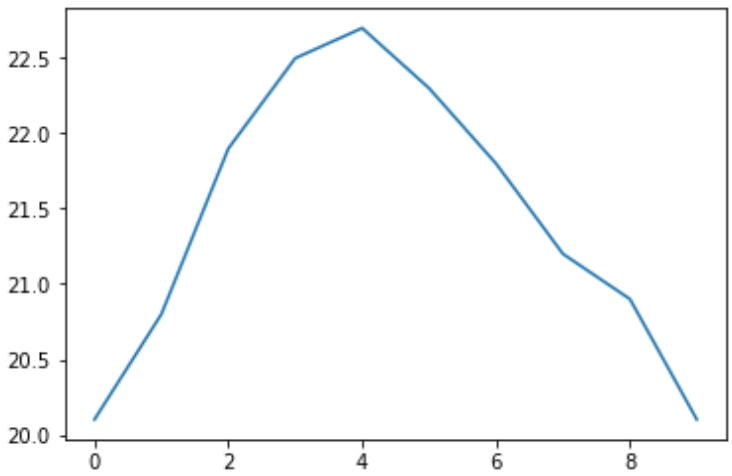
Even though we want to cover the module matplotlib not until a later chapter, we want to demonstrate how we can use this module to depict our temperature values. To do this, we us the package pyplot from matplotlib.

If you use the jupyter notebook, you might be well advised to include the following line of code to prevent an external window to pop up and to have your diagram included in the notebook:

```
%matplotlib inline
```

The code to generate a plot for our values looks like this:

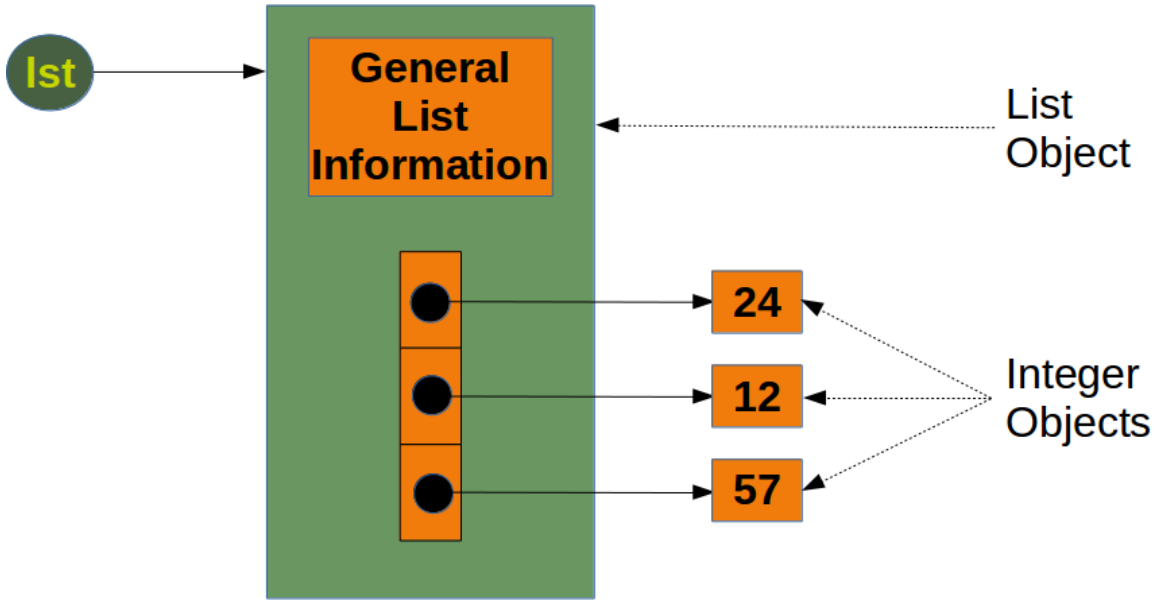
```
import matplotlib.pyplot as plt

plt.plot(C)
plt.show()
```



The function plot uses the values of the array C for the values of the ordinate, i.e. the y-axis. The indices of the array C are taken as values for the abscissa, i.e. the x-axis.

MEMORY CONSUMPTION: NDARRAY AND LIST

The main benefits of using numpy arrays should be smaller memory consumption and better runtime behaviour. We want to look at the memory usage of numpy arrays in this subchapter of our turorial and compare it to the memory consumption of Python lists.



To calculate the memory consumption of the list from the above picture, we will use the function getsizeof from the module sys.

```
from sys import getsizeof as size

lst = [24, 12, 57]

size_of_list_object = size(lst)    # only green box
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57

total_list_size = size_of_list_object + size_of_elements
print("Size without the size of the elements: ", size_of_list_objec
t)
print("Size of all the elements: ", size_of_elements)
print("Total size of list, including elements: ", total_list_size)
```

```
Size without the size of the elements:  96
Size of all the elements:  84
Total size of list, including elements:  180
```

The size of a Python list consists of the general list information, the size needed for the references to the elements and the size of all the elements of the list. If we apply sys.getsizeof to a list, we get only the size without the size of the elements. In the previous example, we made the assumption that all the integer elements of our list have the same size. Of course, this is not valid in general, because memory consumption will be higher for larger integers.

We will check now, how the memory usage changes, if we add another integer element to the list. We also look at an empty list:

```
lst = [24, 12, 57, 42]

size_of_list_object = size(lst)    # only green box
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57, 42

total_list_size = size_of_list_object + size_of_elements
print("Size without the size of the elements: ", size_of_list_objec
t)
print("Size of all the elements: ", size_of_elements)
print("Total size of list, including elements: ", total_list_size)

lst = []
print("Emtpy list size: ", size(lst))
```

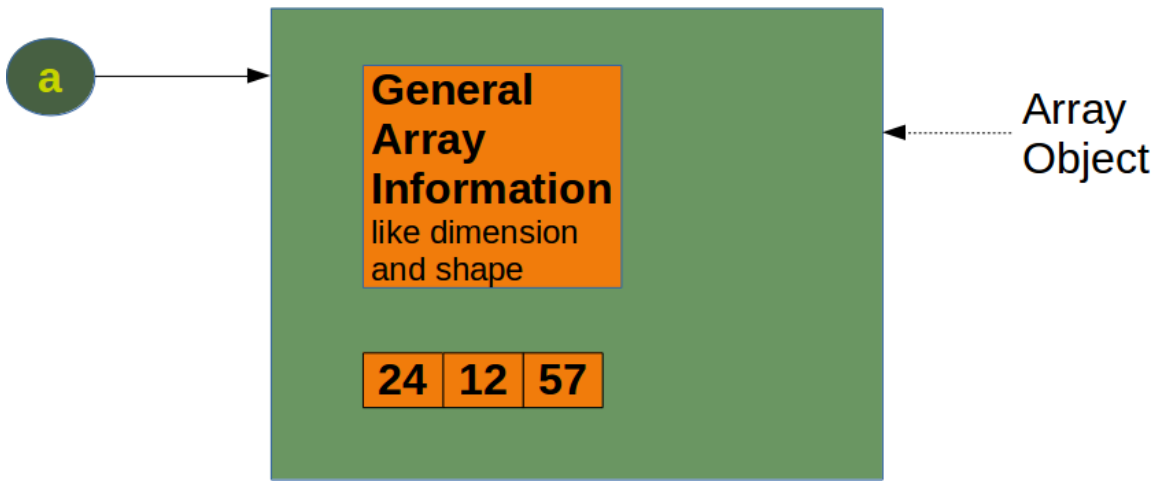
```
Size without the size of the elements:  104
Size of all the elements:  112
Total size of list, including elements:  216
Emtpy list size:  72
```

We can conclude from this that for every new element, we need another eight bytes for the reference to the new object. The new integer object itself consumes 28 bytes. The size of a list "lst" without the size of the elements can be calculated with:

64 + 8 * len(lst)

To get the complete size of an arbitrary list of integers, we have to add the sum of all the sizes of the integers.

We will examine now the memory consumption of a numpy.array. To this purpose, we will have a look at the implementation in the following picture:



We will create the numpy array of the previous diagram and calculate the memory usage:

```
a = np.array([24, 12, 57])
print(size(a))
```

```
120
```

We get the memory usage for the general array information by creating an empty array:

```
e = np.array([])
print(size(e))
```

```
96
```

We can see that the difference between the empty array "e" and the array "a" with three integers consists in 24 Bytes. This means that an arbitrary integer array of length "n" in numpy needs

96 + n * 8 Bytes

whereas a list of integers needs, as we have seen before

64 + 8 *len(lst)* + *len(lst)* 28

This is a minimum estimation, as Python integers can use more than 28 bytes.

When we define a Numpy array, numpy automatically chooses a fixed integer size. In our example "int64". We can determine the size of the integers, when we define an array. Needless to say, this changes the memory requirement:

```
a = np.array([24, 12, 57], np.int8)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int16)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int32)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int64)
print(size(a) - 96)
```

```
3
6
12
24
```

TIME COMPARISON BETWEEN PYTHON LISTS AND NUMPY ARRAYS

One of the main advantages of NumPy is its advantage in time compared to standard Python. Let's look at the following functions:

```
import time
size_of_vec = 1000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = [X[i] + Y[i] for i in range(len(X)) ]
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

Let's call these functions and see the time consumption:

```
t1 = pure_python_version()
t2 = numpy_version()

print(t1, t2)
print("Numpy is in this example " + str(t1/t2) + " faster!")
```

```
0.0010614395141601562 5.2928924560546875e-05
Numpy is in this example 20.054054054054053 faster!
```

It's an easier and above all better way to measure the times by using the timeit module. We will use the Timer class in the following script.

The constructor of a Timer object takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'.

The statements may contain newlines, as long as they don't contain multi-line string literals.

A Timer object has a timeit method. timeit is called with a parameter number:

```
timeit(number=1000000)
```

The main statement will be executed "number" times. This executes the setup statement once, and then returns the time it takes to execute the main statement a "number" of times. It returns the time in seconds.

```
import numpy as np
from timeit import Timer

size_of_vec = 1000

X_list = range(size_of_vec)
Y_list = range(size_of_vec)

X = np.arange(size_of_vec)
Y = np.arange(size_of_vec)

def pure_python_version():
    Z = [X_list[i] + Y_list[i] for i in range(len(X_list)) ]

def numpy_version():
    Z = X + Y


#timer_obj = Timer("x = x + 1", "x = 0")
timer_obj1 = Timer("pure_python_version()",
                   "from __main__ import pure_python_version")
timer_obj2 = Timer("numpy_version()",
                   "from __main__ import numpy_version")

for i in range(3):
    t1 = timer_obj1.timeit(10)
    t2 = timer_obj2.timeit(10)
    print("time for pure Python version: ", t1)
    print("time for Numpy version: ", t2)
    print(f"Numpy was {t1 / t2:7.2f} times faster!")
```

```
time for pure Python version:  0.0021230499987723306
time for Numpy version:  0.0004346180066931993
Numpy was    4.88 times faster!
time for pure Python version:  0.003020321993972175
time for Numpy version:  0.0001488260022597387 4
Numpy was   20.29 times faster!
time for pure Python version:  0.002028984992648475
time for Numpy version:  0.0002098319964716211
Numpy was    9.67 times faster!
```

The repeat() method is a convenience to call timeit() multiple times and return a list of results:

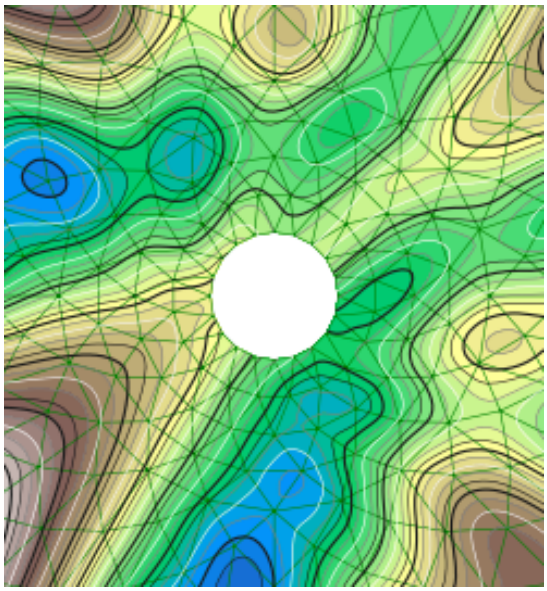
```
print(timer_obj1.repeat(repeat=3, number=10))
print(timer_obj2.repeat(repeat=3, number=10))
```

```
[0.0030275019962573424, 0.002999588003149256, 0.00221208699
80426505]
[6.104000203777105e-05, 0.0001641790004214272, 1.9048005924
56013e-05]
```

In [ ]:

MATPLOTLIB TUTORIAL

INTRODUCTION



Matplotlib is a plotting library like GNUplot. The main advantage towards GNUplot is the fact that Matplotlib is a Python module. Due to the growing interest in python the popularity of matplotlib is continually rising as well.

Another reason for the attractiveness of Matplotlib lies in the fact that it is widely considered to be a perfect alternative to MATLAB, if it is used in combination with Numpy and Scipy. Whereas MATLAB is expensive and closed source, Matplotlib is free and open source code. It is also object-oriented and can be used in an object oriented way. Furthermore it can be used with general-purpose GUI toolkits like wxPython, Qt, and GTK+. There is also a procedural "pylab", which designed to closely resemble that of MATLAB. This can make it extremely easy for MATLAB users to migrate to matplotlib.

Matplotlib can be used to create publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Another characteristic of matplotlib is its steep learning curve, which means that users usually make rapid progress after having started. The officicial website has to say the following about this: "matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code."

A FIRST EXAMPLE

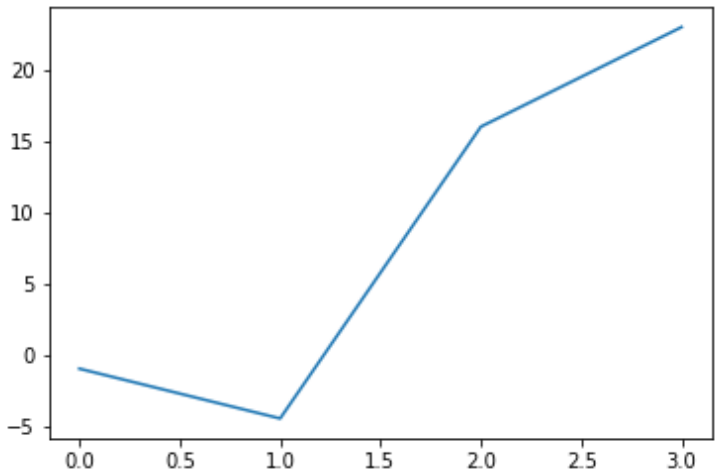
We will start with a simple graph , which is as simple as simple can be. A graph in matplotlib is a two- or three-dimensional drawing showing a relationship by means of points, a curve, or amongst others a series of bars. We have two axis: The horizontal X-axis is representing the independent values and the vertical Y-axis corresponds to the depended values.

We will use the pyplot submodule of matplotlib. pyplot provides a procedural interface to the object-oriented plotting library of matplotlib.
Its plotting commands are chosen in a way that they are similar to Matlab both in naming and with the arguments.

Is is common practice to rename matplotlib.pyplot to plt. We will use the plot function of pyplot in our first example. We will pass a list of values to the plot function. Plot takes these as Y values. The indices of the list are automatically taken as the X values. The command `%matplotlib inline` makes only sense, if you work with Ipython Notebook. It makes sure, that the graphs will be depicted inside of the document and not as independent windows:

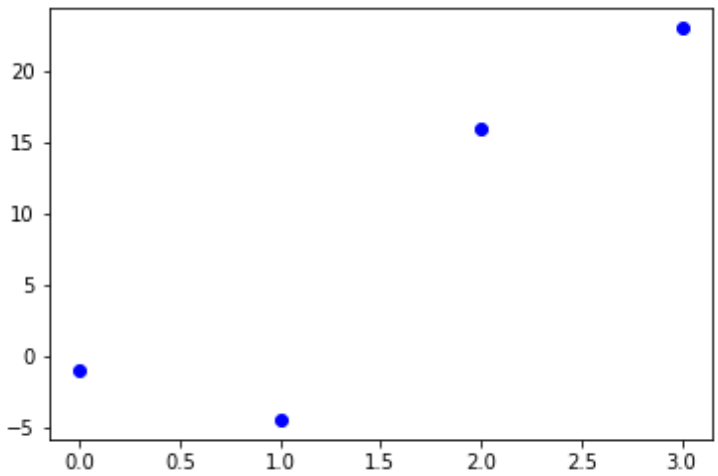
```
%matplotlib inline
```

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23])
plt.show()
```



What we see is a continuous graph, even though we provided discrete data for the Y values. By adding a format string to the function call of plot, we can create a graph with discrete values, in our case blue circle markers. The format string defines the way how the discrete points have to be rendered.

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23], "ob")
plt.show()
```



THE FORMAT PARAMETER OF PYPLOT.PLOT

We have used "ob" in our previous example as the format parameter. It consists of two characters. The first one defines the line style or the dicrete value style, i.e. the markers, while the second one chooses a colour for the graph. The order of the two characters could have been reversed, i.e. we could have written it as "bo" as well. If the format parameter is not given, as in the first example, the default value is "b-", i.e. a solid blue line.

The following format string characters are accepted to control the line style or marker:

```
============================================
character       description
============================================
'-'             solid line style
'--'            dashed line style
'-.'            dash-dot line style
':'             dotted line style
'.'             point marker
','             pixel marker
'o'             circle marker
'v'             triangle_down marker
'^'             triangle_up marker
'<'             triangle_left marker
'>'             triangle_right marker
'1'             tri_down marker
'2'             tri_up marker
'3'             tri_left marker
'4'             tri_right marker
's'             square marker
'p'             pentagon marker
'*'             star marker
'h'             hexagon1 marker
'H'             hexagon2 marker
'+'             plus marker
'x'             x marker
'D'             diamond marker
'd'             thin_diamond marker
'|'             vline marker
'_'             hline marker
============================================
```

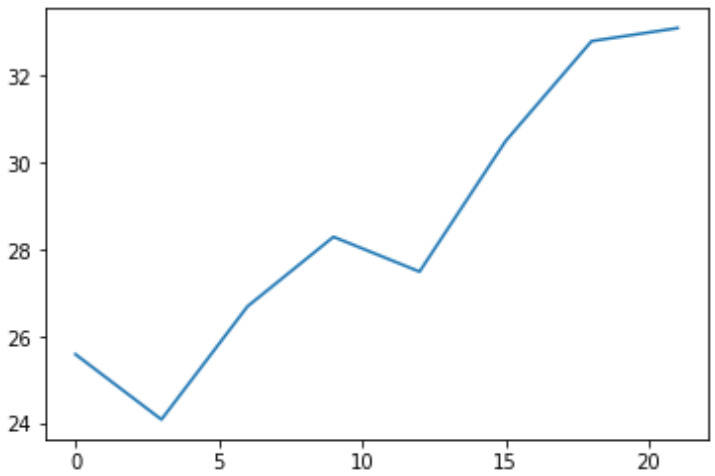
The following color abbreviations are supported:

```
==================
character   color
==================
'b'         blue
'g'         green
'r'         red
'c'         cyan
'm'         magenta
'y'         yellow
'k'         black
'w'         white
==================
```

As you may have guessed already, you can add X values to the plot function. We will use the multiples of 3 starting at 3 below 22 as the X values of the plot in the following example:

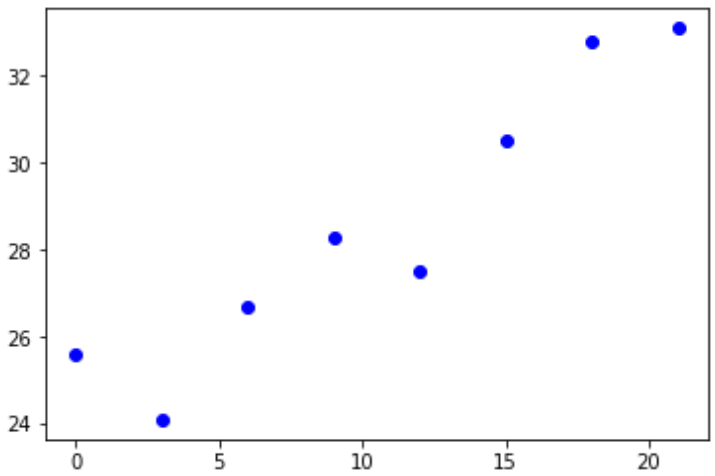
```
import matplotlib.pyplot as plt

# our X values:
days = list(range(0, 22, 3))
print("Values of days:", days)
# our Y values:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

plt.plot(days, celsius_values)
plt.show()
```

```
Values of days: [0, 3, 6, 9, 12, 15, 18, 21]
```



... and once more with discrete values:

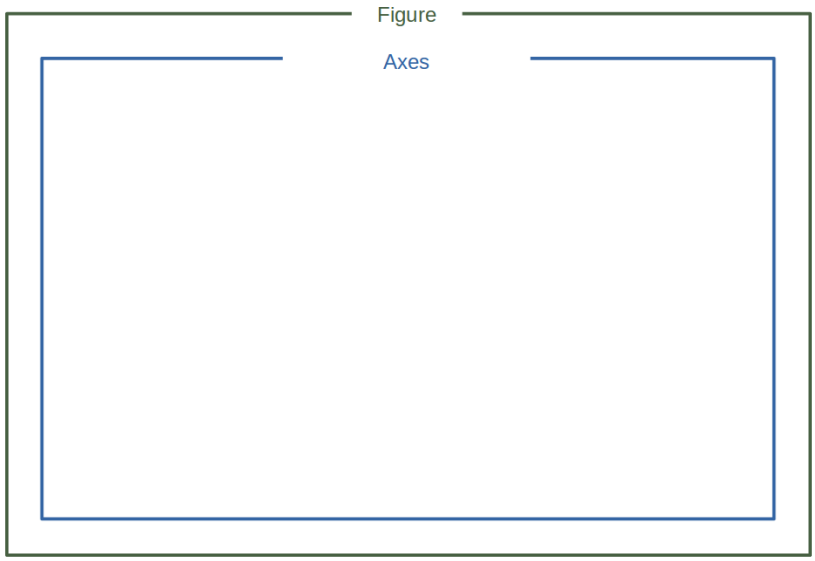
```
plt.plot(days, celsius_values, 'bo')
plt.show()
```



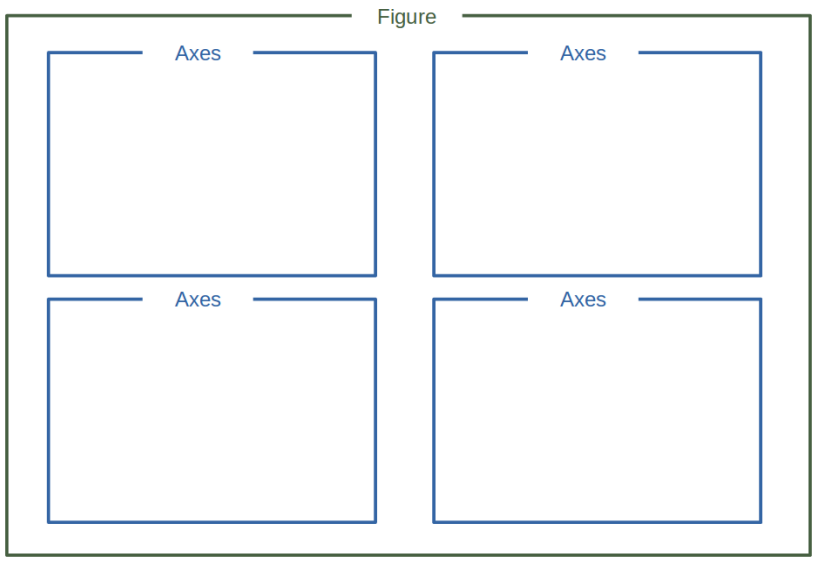
MATPLOTLIB OBJECT HIERARCHY AND IMPORTANT TERMS

Matplotlib is hierarchically organized. In the previous example you couldn't see see, because we haven't made use of this structure. All we used was a simple plot which is implicitly building the necessary structures.

The top of the tree-like structure of matplotlib objects is the `figure` object. A figure can be seen as the container which contains one or more plots. The plots are called `axes` in matplotlib jargon. The following diagram shows a figure with one axes:



As we have already mentioned a figure may also contain more than one axes:



The terms `axes` and `figure` and in particular the relationship between them can be quite confusing to beginners in Matplotlib. Similarly difficult to access and understand are many of the terms `Spine`, `Tick` and `Axis`. Their function and meaning are easy to crasp, if you see look at them in the following diagram:

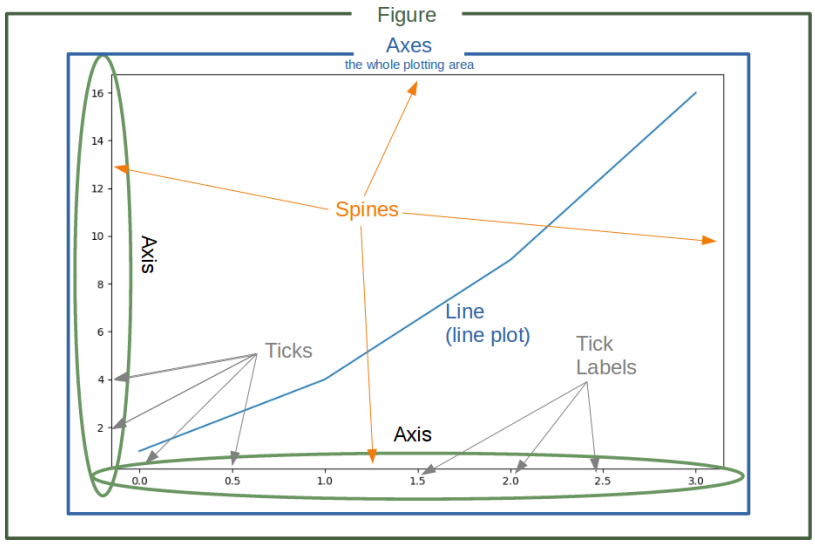
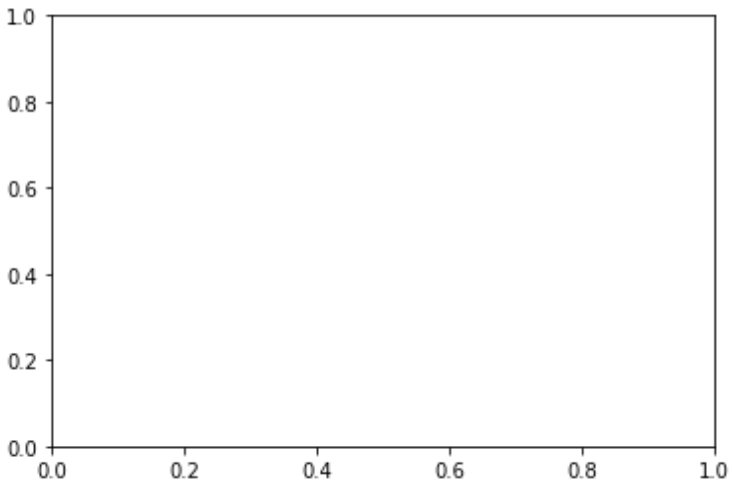


FIGURE AND AXES ERZEUGEN

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

print(type(fig))
print(type(ax))
```

```
<class 'matplotlib.figure.Figure'>
<class 'matplotlib.axes._subplots.AxesSubplot'>
```



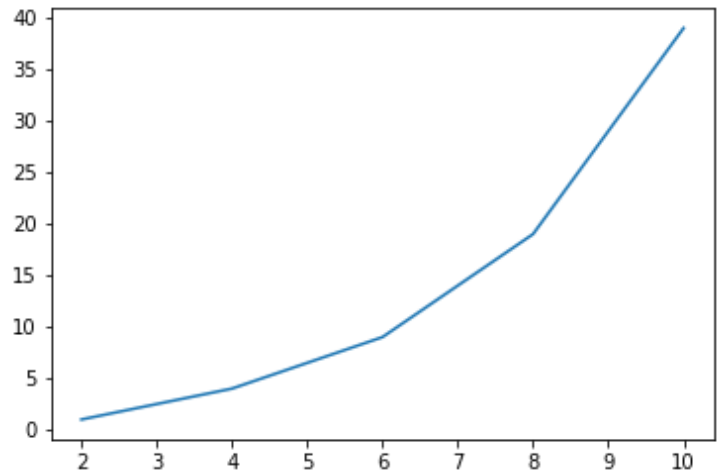
The function `subplots` can be used to create a figure and a set of subplots. In our previous example, we called the function without parameters which creates a figure with a single included axes. we will see later how to create more than one axes with `subplots` .

We will demonstrate in the following example, how we can go on with these objects to create a plot. We can see that we apply `plot` directly on the axis object `ax` . This leaves no possible ambiguity as in the `plt.plot` approach:

```
import matplotlib.pyplot as plt

# Data for plotting
X = [2, 4, 6, 8, 10]
Y = [1, 4, 9, 19, 39]

fig, ax = plt.subplots()
ax.plot(X, Y)
```

Output: :

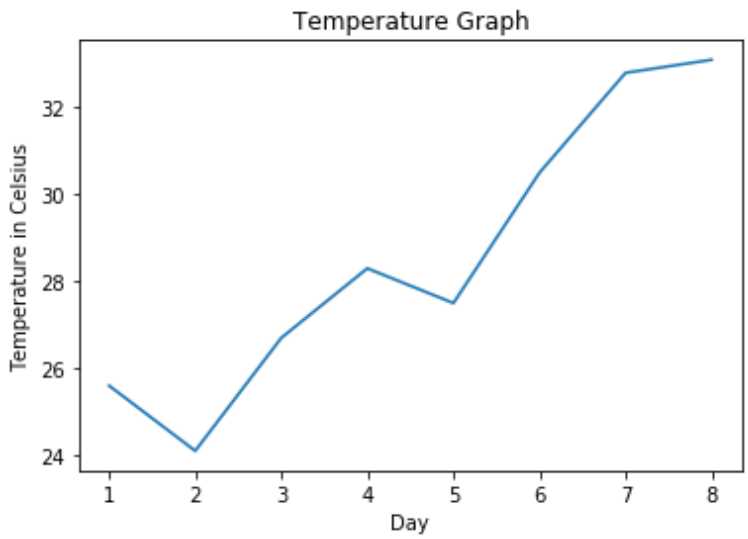
```
[<matplotlib.lines.Line2D at 0x7f5dbe7b6dd8>]
```



LABELS ON AXES

We can improve the appearance of our graph by adding labels to the axes. We also want to give our plot a headline or let us call it a `title` to stay in the terminology of Matplotlib. We can accomplish this by using the `set` method of the axis object `ax` :

```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

fig, ax = plt.subplots()
ax.plot(days, celsius_values)
ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

plt.show()
```

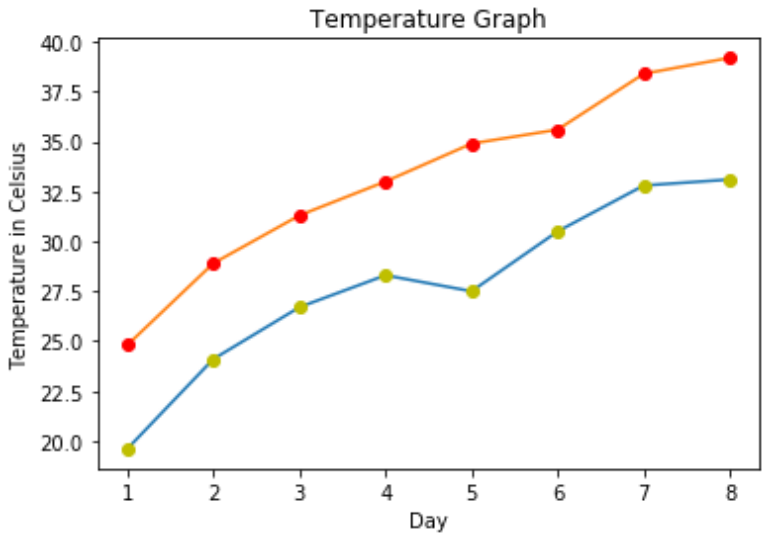


THE PLOT FUNCTION

The `plot` function is needed to plot a figure or better the figures, because there may be more than one. When plot is run in ipython with its pylab mode, it displays all figures and returns to the ipython prompt. When we run it in non-interactive mode, - which is the case, when we run a Python program - it displays all figures and blocks until the figures have been closed.

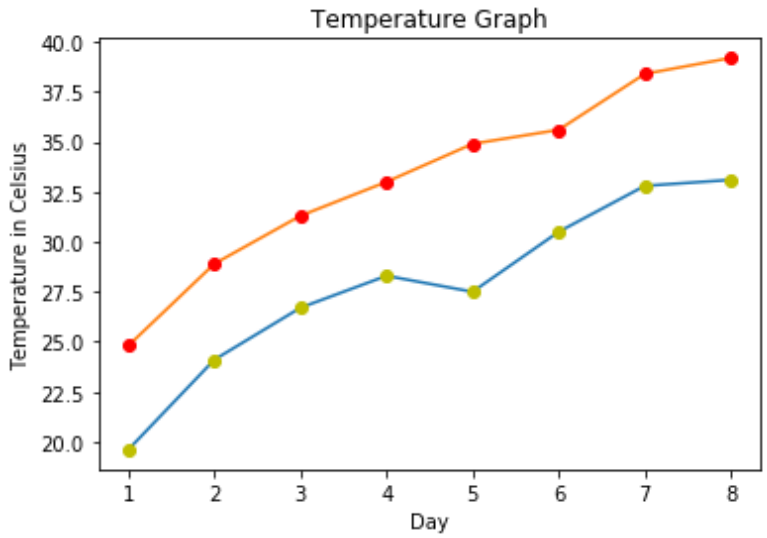
We can specify an arbitrary number of x, y, fmt groups in a plot function. We will extend our previous temperature example to demonstrate this. We provide two lists with temperature values, one for the minimum and one for the maximum values:

```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')


ax.plot(days, celsius_min,
        days, celsius_min, "oy",
        days, celsius_max,
        days, celsius_max, "or")

plt.show()
```



We could have used for plot calls in the previous code instead of one, even though this is not very attractive:

```
import matplotlib.pyplot as plt

days = list(range(1, 9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

ax.plot(days, celsius_min)
ax.plot(days, celsius_min, "oy")
ax.plot(days, celsius_max)
ax.plot(days, celsius_max, "or")

plt.show()
```

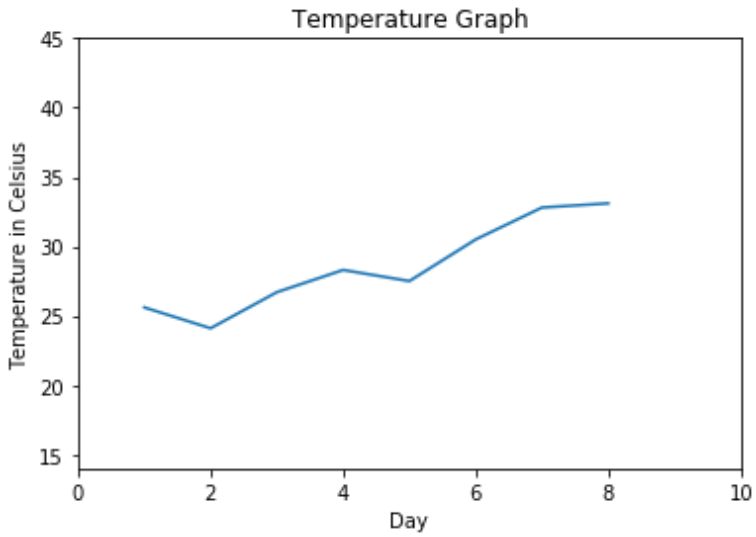


CHECKING AND DEFINING THE RANGE OF AXES

We can also check and define the range of the axes with the function axis. If you call it without arguments it returns the current axis limits:

```
import matplotlib.pyplot as plt

days = list(range(1, 9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

fig, ax = plt.subplots()
ax.plot(days, celsius_values)
ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

print("The current limits for the axes are:")
print(ax.axis())
print("We set the axes to the following values:")
xmin, xmax, ymin, ymax = 0, 10, 14, 45
print(xmin, xmax, ymin, ymax)
ax.axis([xmin, xmax, ymin, ymax])
plt.show()
```

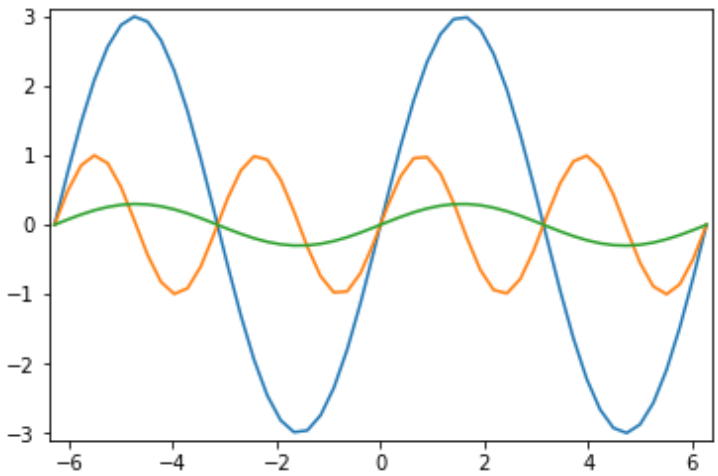
```
The current limits for the axes are:
(0.6499999999999999, 8.35, 23.650000000000002, 33.550000000
000004)
We set the axes to the following values:
0 10 14 45
```



"LINSPACE" TO DEFINE X VALUES

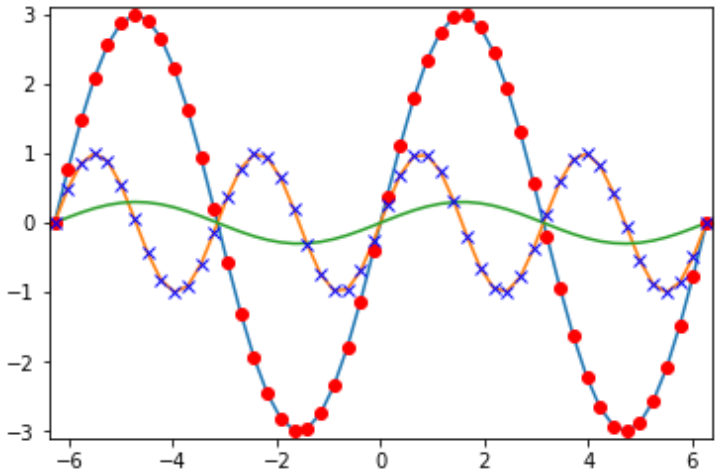
We will use the Numpy function linspace in the following example. linspace can be used to create evenly spaced numbers over a specified interval.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)

fig, ax = plt.subplots()

startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1
ax.axis([startx, endx, starty, endy])

ax.plot(X, F1, X, F2, X, F3)
plt.show()
```



The next example isn't anything new. We will just add two more plots with discrete points:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)

fig, ax = plt.subplots()
startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1

ax.axis([startx, endx, starty, endy])
ax.plot(X, F1, X, F2, X, F3)
ax.plot(X, F1, 'ro', X, F2, 'bx')
plt.show()
```



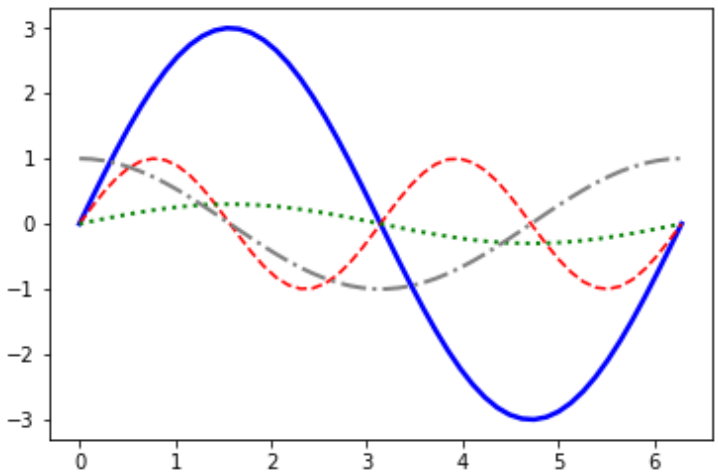
CHANGING THE LINE STYLE

The linestyle of a plot can be influenced with the linestyle or ls parameter of the plot function. It can be set to one of the following values:

`'-', '--', '-.', ':', 'None', ' ', ''`

We can use linewidth to set the width of a line as the name implies.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
F4 = np.cos(X)

fig, ax = plt.subplots()
ax.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
ax.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
ax.plot(X, F3, color="green", linewidth=2, linestyle=":")
ax.plot(X, F4, color="grey", linewidth=2, linestyle="-.")
plt.show()
```



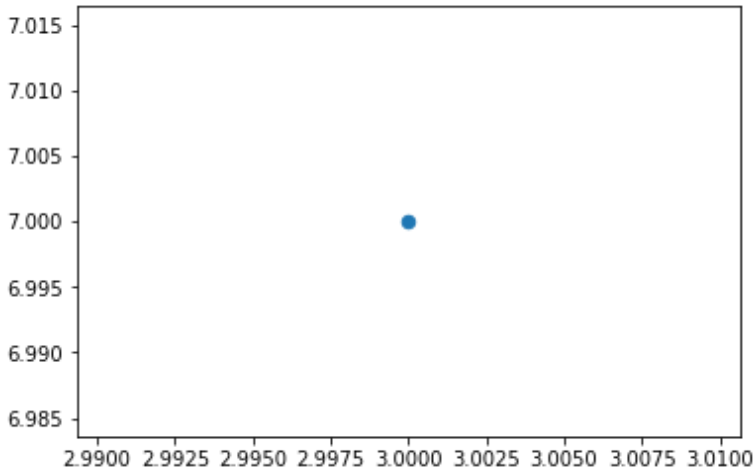
DRAW LPOINTS IN MATPLOTLIB

We will learn now how to draw single points in Matplotlib.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(3, 7, s=42)
```

Output: :

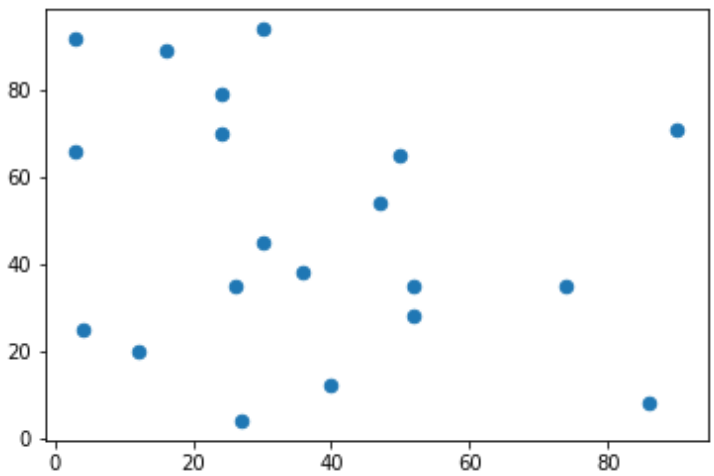
```
<matplotlib.collections.PathCollection at 0x7f5dbe7fc8d0>
```




```
import matplotlib.pyplot as plt
import numpy as np

X = np.random.randint(0, 100, (20,))
Y = np.random.randint(0, 100, (20,))
fig, ax = plt.subplots()
ax.scatter(X, Y, s=42)
```

Output: :

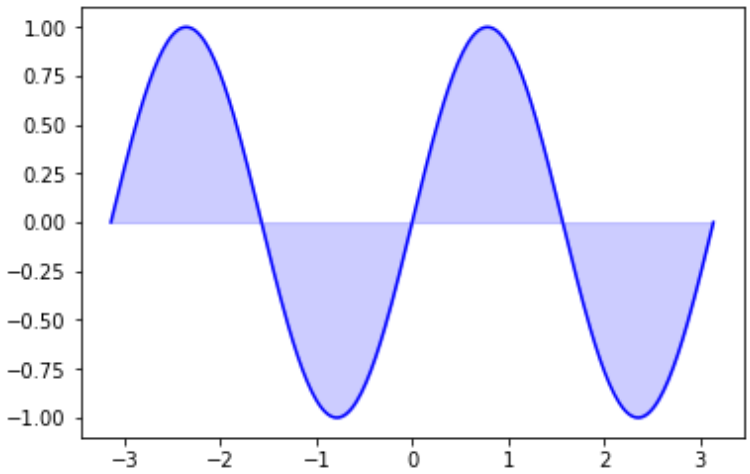
```
<matplotlib.collections.PathCollection at 0x7f5dbe518a90>
```



SHADING REGIONS WITH FILL_BETWEEN()

It is possible to shade or colorize regions between two curves. We are filling the region between the X axis and the graph of `sin(2*X)` in the following example:

```
import numpy as np
import matplotlib.pyplot as plt
n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

fig, ax = plt.subplots()
ax.plot (X, Y, color='blue', alpha=1.0)
ax.fill_between(X, 0, Y, color='blue', alpha=.2)
plt.show()
```



The general syntax of fill_between:

```
fill_between(x, y1, y2=0, where=None, interpolate=False, **kwargs)
```

The parameters of fill_between:

Parameter	Meaning
x	An N-length array of the x data
y1	An N-length array (or scalar) of the y data
y2	An N-length array (or scalar) of the y data
where	If None, default to fill between everywhere. If not None, it is an N-length numpy boolean array and the fill will only happen over the regions where where==True.
interpolate	If True, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the x array.
kwargs	Keyword args passed on to the PolyCollection

```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

fig, ax = plt.subplots()

ax.plot (X, Y, color='blue', alpha=1.00)
ax.fill_between(X, Y, 1, color='blue', alpha=.1)
plt.show()
```