# Python 101 Documentation

***Release 0.1.0***

**Dominik Steinberger**

**Oct 20, 2016**

> "For the things we have to learn before we can do them, we learn by doing them."
>
> —Aristotle (around 350 BCE)

Welcome to the Python tutorial of the Institute of Materials Simulation. The goal of this tutorial is to introduce the basic tools used in our everyday scientific worklife by actually performing such tasks. We will start with a brief introduction to the *Unix Shell* in which we list the most common commands and what they do.

## Contents

CHAPTER 1

# The Unix Shell

> "The Linux philosophy is 'Laugh in the face of danger'. Oops. Wrong One. 'Do it yourself'. Yes, that's it."
>
> —Linus Torvalds

We could try to think of a neat way of defining what a *Unix shell* is, but thankfully Wikipedia comes to the rescue:

> A Unix shell is a command-line interpreter or shell that provides a traditional Unix-like command line user interface. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands.

To put it in simpler terms, instead of pointing and clicking on colorful icons to perform actions like opening a directory or a program, you type commands in a terminal. The most commonly used commands are outlined in the following sections.

---

**Note:** If you do not have any prior experience with command line interfaces (CLIs), working with one might seem daunting at first. This is perfectly normal. It takes time to get used to the Unix shell. The key to mastering the Unix shell is to use it. *A lot.*

---

## 1.1 `pwd`

This command is used to show the path to the current working directory. It is usually used to see where you currently are.

```
$ pwd
```

## 1.2 `ls`

This command is used to show the contents of the directory you specify as argument.

```
$ ls directory1
```

If you do not provide anything besides `ls` it prints the contents of the current working directory.

```
$ ls
```

For a more verbose version you may use the `-l` options:

```
$ ls -l
```

**Note:** If you have some spaces in some of your file names you have to enclose the whole filename like this: `"{filename}"`

## 1.3 cd

This command is used to change the directory to the one specified as argument. If no argument is provided you are changing to your home directory.

```
$ cd directory1
```

To get to the parent directory use

```
$ cd ..
```

**Note:** In a lot of cases `bash` – the shell you are using – provides rather intelligent *autocompletion*. To use this start typing the name of a file or directory and hit the `tab` button. If there is a unique completion option it is completed automatically. Else hitting tab another time will give you a list of options that starts with whatever you typed until then.

## 1.4 cat

This command is used to print the contents of the files specified as arguments.

```
$ cat file1 file2 ... fileN
```

## 1.5 cp

This command is used to copy files. For example

```
$ cp file1 file2
```

copies `file1` to `file2`. If you want to copy a lot of files to another directory use

```
$ cp file1 file2 ... fileN directory1/
```

Copying a whole directory requires you to use the `-r` option:

```
$ cp -r directory1 directory2
```

## 1.6 mv

This command is used to move files. For example

```
$ mv file1 file2
```

essentially renames `file1` to `file2`. To move several files into a directory use

```
$ mv file1 file2 ... fileN directory1/
```

As opposed to `cp` the `mv` command can move whole directories without using the `-r` option:

```

```

## 1.7 touch

This command is used to create an empty file. Using

```
$ touch file1
```

hence results in an empty file with the name `file1`.

**Note:** If you want to copy something from the Terminal you can not do this via the key combination `Ctrl + C` as this is reserved for cancelling the running program. Instead use `Ctrl + Shift + C`. For pasting you also have to use `Ctrl + Shift + V`.

## 1.8 rm

This command is used to delete files and directories. Hence

```
$ rm file1
```

deletes `file1`.

**Warning:** If you delete files or directories on a modern, graphical operating system the files and directories usually do not get deleted immediately, but are copied to an intermediate directory that is usually called `trash bin`. This could be considered a safety measure against accidentally deleting important files. This "safety net" does not exist for the `rm` command. Whatever you delete via `rm` is permanently deleted.

## 1.9 Summary

**pwd** Print the path to the directory you are currently in.

**ls $1** List the contents of directory specified by `$1`. If you do not specify a directory it defaults to your current directory.

**cd $1** Change the directory to `$1`. If you do not specify a directory you go to your home directory. If you want to go back to your last directory you can use `cd -`.

**cat $1 $2 ... $n** Read the files specified and print their content to the terminal.

**cp $1 $2** Copy the first argument to the second argument. If you want to copy a directory you have to use it with the `-r` option: `cp -r $1 $2`.

**mv $1 $2** Move the first argument to the second argument. It is basically like renaming the first argument.

**touch $1** Create an empty file at `$1`.

**rm $1 $2 ... $n** Delete the files specified. If you want to delete a directory and its contents you have to use it with the `-r` option: `rm -r $1`.

## 1.10 Tasks

1. Create an empty file called `my_first_file.txt`
2. Open the file with your text editor and fill it with something other and `asdf`. Save and close afterwards.
3. Print the content of the file to the terminal.
4. Make a new directory named `my_first_folder`
5. Copy the file `my_first_file.txt` into this directory.
6. Remove the old file.
7. Print the content of the file `my_first_file.txt` in the directory `my_first_directory` to the terminal.
8. Print your current working directory.
9. Enter the directory `my_first_directory`.
10. Print your current working directory.
11. Enter the parent directory.
12. Delete the directory `my_first_directory`.

CHAPTER 2

# Installing Python

How to install Python depends a lot on your operating system.

## 2.1 macOS

1. Install the current version of Xcode on your Mac via the Mac App Store.
2. Open your terminal and install the Xcode Command Line Utilities and run

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

3. Agree to the Xcode license by running

```
sudo xcodebuild -license
```

4. Install MacPorts for your version of macOS.
5. Install the required ports via

```
sudo port install freetype libpng git hdf5 python35
```

## 2.2 Ubuntu 16.04 LTS

Install the required packages via

```
sudo apt-get install cmake gfortran git libfreetype6-dev libatlas-dev liblapack-dev␣
↪libhdf5-dev python3-dev python3-venv python3-pip python3-tk
```

CHAPTER 3

# Basics

> "The canonical, 'Python is a great first language', elicited, 'Python is a great last language!'"
>
> —Noah Spurrier

Now we are going to introduce the basics of the Python programming language. We start with the infamous *Hello, World!* program and the basic syntax of a Python script.

**Attention:** Make sure you have your virtual environment activated! If you do not have `(python101)` in front of your command line prompt you need to activate it using

```
$ source ~/.virtualenvs/python101/bin/activate
```

Execute the following command to start the interactive Python interpreter:

```
$ python
```

You should see a couple of lines printed in the terminal, with the first line stating, among the current date and time, the version of Python you use. The very last line should start with >>> which indicates the Python prompt. You can write Python commands there and execute them.

**Attention:** In the previous chapter *The Unix Shell* the basic commands for the Unix shell were introduced. Notice how every command was preceeded by a `$` character. In this tutorial code blocks that start with a `$` sign are meant to be executed in the Unix shell. If the code block is preceeded by >>> it means that it should either be executed in the Python interpreter or be used in a Python script.

## 3.1 Python as an Interactive Calculator

To get your feet wet with Python you can use the Python interpreter as a calculator. You have the usual mathematical operators at your disposal, like

- `+` addition,
- `-` subtraction,
- `*` multiplication,
- `/` division,
- `**` exponent,

`//` integer division, and

`%` modulus.

If you are not familiar with one of them just give it a try in the Python interpreter—do not only use integers, but also floating point numbers. You can also use brackets as you would use them in mathematical expressions. Try whether Python uses the proper mathematical rules with regards to the order of execution of the operators.

---

**Note:** At one point you will enter an "invalid" expression like `1 + * 2`. Python will then raise a `SyntaxError` to tell you that whatever you typed is not valid Python syntax. In many cases Python will also give you additional information about the error. There are many more errors you can encounter, and it is perfectly normal to have errors. The only difference between a seasoned programmer and a beginner is the time it takes to fix those errors. The more errors and mistakes you made the better you know how to solve them.

---

To leave the Python interpreter you either execute

```
>>> quit()
```

or you press `Ctrl + D`. Besides the interactive Python interpreter you can also write scripts with Python. Scripts are files that can be executed from the command line interface. They contain Python expressions that get executed once you call the scripts. A script can be simple and merely rename files or it can be complex and run a complete simulation of a car crash. You decide how simple they are.

## 3.2 Your First Python Script

We will start with the infamous Hello, World! Program. Open a new terminal, activate your virtual environment, and create a new file named `hello_world.py` via

```
$ touch hello_world.py
```

Open it with your favorite text editor, e.g., *Atom* or *SublimeText*. In the former case you would open the file via

```
$ atom hello_world.py
```

Now type (not copy!) the following into the file:

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

Save the file, switch to your command line interface, and execute

```
$ python hello_world.py
```

If you did everything correctly you should see the phrase `Hello,World!` popping up in your command line interface. If you see something like

```
  File "hello_world.py", line 1
    print('Hello, World!)
                         ^
SyntaxError: EOL while scanning string literal
```

or

```
  File "hello_world.py", line 2

```

```
                          ^
SyntaxError: unexpected EOF while parsing
```

it means that you have either forgotten the closing `'` or `)`, respectively. As you can see Python tries its best to describe the error to you so that it can be fixed quickly.

If everything went fine: **Congratulations!** You wrote your first Python script!

## 3.3 The `print` Function

The function you used in your first Python script, the `print()` function, has a rather simple goal: Take whatever you have in there and display it in the command line interface. In the Python interpreter (the command line starting with `>>>`) the result of an expression was displayed automatically. Try creating a new file `math_expressions.py` and enter several mathematical expressions like you did earlier. Save the file, switch to your terminal and execute the file via

```
python math_expressions.py
```

You should not see a single thing happening. That is because you never told Python what to actually do with those expressions. So what it does is evaluate them and nothing more. Now wrap the mathematical expressions in the `print()` function, for example like this:

```
print((3 + 4)*6)
```

If you execute the script again you should see the expected output.

## 3.4 Integers, Floats and Strings

In the previous examples you worked with integers, floating-point numbers, and with strings. `-4`, `0`, and `2` are all integers. `1.2`, `1.0` and `-2e2` (which is the scientific notation for `-200.0`) are floating-point numbers. Finally, `'Hello,World!'` is a string. These categories are called data types. Every value in Python is of a certain data type.

The meaning of operators may depend on the data types of the values surrounding it. Take, e.g., the addition operator `+`:

```
>>> 1 + 2
3
>>> 1.2 + 3.4
4.6
>>> 'My first sentence.' + 'My second sentence.'
'My first sentence.My second sentence.'
>>> 'My ' + 3 + 'rd sentence.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

In the last case the addition operator has no idea how to combine the integer `1` with the strings. What you can do to solve this is to convert the integer to a string using `str()`:

```
>>> 'My ' + str(3) + 'rd sentence.'
'My 3rd sentence.'
```

If you want to convert something to a string you use `str()`, to convert to an integer you use `int()`, for floating-point numbers you use `float()`.

```
>>> '1.2' + '3.4'
'1.23.4'
>>> float('1.2') + float('3.4')
4.6
>>> int('1.4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.4'
>>> str(1e2)
'100.0'
```

Play around with those three functions to see what can be converted and what can not. Try the different operators, e.g., try to multiply a string with an integer, etc.

## 3.5 Variables

Like in mathematics you can also use variables to store values. A variable has a name by which it is called and a value. There are three rules that a variable name must comply:

1. It must be exactly one word.
2. It must comprise only letters, numbers, and the underscore character.
3. It must not begin with a number.

Other than that anything goes. To assign a value to a variable you use the equal sign = with the variable name on the left and the value on the right:

```
>>> my_first_variable = 21
>>> 2*my_first_variable
42
>>> my_second_variable = 3
>>> my_first_variable/my_second_variable
7.0
>>> my_third_variable = my_first_variable
>>> print(my_third_variable)
21
```

Here is a slightly more complex example:

```
students = 35
tutors = 2
classrooms = 1
pizza_orders = 20

students_per_tutor = students / tutors
persons = students + tutors
persons_per_classroom = persons / classrooms
hungry_persons = persons - pizza_orders

print('There are', students, 'students and', tutors, 'tutors.')
print('That makes', persons, 'persons in', classrooms, 'class room(s).')
print(hungry_persons, 'have to stay hungry...')
```

The advantages of using variables are two-fold:

- If the amount of students, tutors, classrooms or pizza_orders changes you only have to update one line instead of many. This is less error-prone and faster.
- You give the values some meaning which should be represented in the variable name. You could in principle read "the students per tutor is the amount of students divided by the amount of tutors." This makes your code easily comprehensible and you need fewer comments. But you still should write them when they make sense!

And here is what the output should look like:

```
There are 35 students and 2 tutors.
That makes 37 persons in 1 class room(s).
17 have to stay hungry...
```

Notice how we used `,` to separate strings and variable names in `print()`, but everything was composed in a nice way? The reason for this is that `print()` can take an arbitrary amount of arguments. Just chain them using `,` and you are good to go. How this works is part of the section *Functions*.

## 3.6 User Input

In some cases you may want to ask the user of your script to provide some additional information, like the path to a file or parameters for a simulation. For this the `input()` can be used.

```
print('What is your name?')
name = input()
print('Nice to meet you,', name)
```

---

**Note:** The value returned by `input()` is always a string. So when you are asking for numbers you have to convert them.

```
print('What is your age in years?')
age = int(input())
print('In 5 years you will be', age+5, 'years old.')
```

---

## 3.7 Imports

Sometimes the features that Python offers by default are not enough. What if you want to use the $\sin(x)$ function? For more specializied topics Python offers modules or packages, either ones that already ship with every Python installation or packages from external parties. The packages that Python ships with are called the standard library. External packages may be, e.g., NumPy and SciPy for scientific computing with Python, or Matplotlib for plotting.

You activate this additional functionality by *importing* these packages in your script:

```
>>> import math
```

Now we have access to all functions available in the `math` module.

```
>>> math.pi
3.141592653589793
>>> math.sin(0.5*math.pi)
1.0
```

Take your time and browse the documentation of the `math` module, try some of the provided functions like `math.ceil()`, `math.exp()`, etc.

## 3.8 Summary

- You can use the interactive Python interpreter to execute small commands.
- You can execute scripts that hold several commands using Python.
- You can display results of computations or strings using the `print()` function
- You can use `str()`, `int()`, `float()` to convert from one data type to another—if it is somehow possible.
- You can store values in variables to access them at a later point in your script.
- You can import modules or packages to extend Pythons builtin functionality using the import statement.

## 3.9 Exercises

1. Write a script that asks the user for the radius of a circle and subsequently shows the circumference and the area of the circle in the terminal.

CHAPTER 4

# Comments

Once your programs grow longer and the logic gets more and more complicated it is a *very* good idea to place *meaningful* comments in your code. This makes it easier for the future you and your collaborators or supervisors to actually understand why your implementation is as it is.

In the very beginning your comments may be used to outline what the code block following the comment is actually doing. Once you get more familiar with Python the content of the comment should change from *what* the code is doing to *why* it is doing it.

To actually write a comment prepend it with the # character. So if you write

```
# In the following example you should only see
# ``Now you see me.`` in the terminal.

print('Now you see me.')
# print("Now you don't.")
print('Are you still seeing me?')  # Inline comments are nice as well!
```

into a file and execute it you should only see

```
Now you see me.
Are you still seeing me?
```

**Note:** In all the python files provided in this tutorial you will see

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

as the first line. As it is a comment it should not have any special meaning, but it does. This line tells the Python interpreter that the source code is written with the UTF-8 character encoding. It does not actively do something in your code, though.

## 4.1 Summary

- You can use comments by prepending #
- Comments are useful to explain why you did something the way you did it

## 4.2 Exercises

1. If you encounter some more complicated code-blocks in the future comment them. Admittedly this is not directly an exercise, but a sincere request.
2. Really, comment your code for others and your future self. You will forget why you coded something the way you did. Still not really an exercise.

CHAPTER 5

# Containers

At one point you may want to keep objects like strings or numbers in another object to show that they are strongly related. These objects that contain other objects are called *container*. Python offers a fair amount of useful containers out of the box, some of which are introduced in the following.

## 5.1 Lists

If you need a flexible container that can take in about anything (even itself) and you need to keep an order to the contained objects you should use a `list`. An empty list is generated by either

```
>>> list()
[]
```

or

```
>>> []
[]
```

Or you generate a non-empty list

```
>>> list([1, 2, 3])
[1, 2, 3]
```

or

```
>>> [1, 2, 3]
[1, 2, 3]
```

Accessing values for the list is done like this:

```
>>> x = [1, 2, 3]
>>> print(x[1])
2
```

**Note:** As you can see in the example above specifying x[1] did not give you the first element of the list, but the second. This is due to Python starting indexing at 0!

With this you can also change values in a list

```
>>> x = [1, 2, 3]
>>> x[1] = 4
>>> print(x)
[1, 4, 3]
```

Negative indices can be used to retrieve element from the end:

```
>>> x = [1, 2, 3, 4, 5]
>>> print(x[-1])
5
>>> print(x[-2])
4
```

You can also create slices of the list using the following notation:

```
list[start:end:step]
```

Its usage is best made clear by some examples:

```
>>> x = ['zero', 'one', 'two', 'three', 'four']
>>> print(x[2:])
['two', 'three', 'four']
>>> print(x[:3])
['zero', 'one', 'two']
>>> print(x[1:3])
['one', 'two']
>>> print(x[::2])
['zero', 'two', 'four']
>>> print(x[::-1])
['four', 'three', 'two', 'one', 'zero']
>>> print(x[-2:])
['three', 'four']
```

You can also append values to a list

```
>>> x = [1, 2, 3]
>>> print(x)
[1, 2, 3]
>>> x.append(4)
>>> print(x)
[1, 2, 3, 4]
```

or insert values at specified positions

```
>>> x = [1, 2, 3]
>>> print(x)
[1, 2, 3]
>>> x.insert(0, -1)
>>> print(x)
[-1, 1, 2, 3]
>>> x.insert(1, 0)
>>> print(x)
[-1, 0, 1, 2, 3]
```

A list can hold anything, even other lists

```
>>> [1, 'two', ['three', 4]]
[1, 'two', ['three', 4]]
```

## 5.2 Tuples

If you are sure that you do not want to modify the data container after its generation and want to keep everything in the order you specified the `tuple` container is the container of choice. They may be initialized by

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> (1, 2, 3)
(1, 2, 3)
```

Accessing single elements is done like in lists:

```
>>> x = (1, 2, 3)
>>> print(x[1])
2
```

And when you try to modify them an error is raised:

```
>>> x = (1, 2, 3)
>>> x[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## 5.3 Sets

If you do not want duplicates in your container and you do not care about their order you should use a `set`. They are initialized via

```
>>> set()  # This one is empty
set()
>>> set([1, 2, 2, 3, 3, 'three'])
{1, 2, 3, 'three'}
>>> {1, 2, 2, 3, 3, 'three'}
{1, 'three', 2, 3}
```

and elements can be added like this:

```
>>> x = {1, 2, 3}
>>> print(x)
{1, 2, 3}
>>> x.add('four')
>>> print(x)
{1, 2, 'four', 3}
```

`Sets` furthermore support a lot of the operators you are familiar with from the set in mathematics. For further information see the Python documentation.

## 5.4 Dictionaries

If you do not want to keep the order of things but would rather like to use keys to access the objects in the container the `dict` has got you covered. `Dicts` are initialized via

```
>>> dict()
{}
>>> {}
{}
>>> x = {'One': 1, 'two': 2, 'THREE': ['A', 'list', 'of', 'strings']}
{'THREE': ['A', 'list', 'of', 'strings'], 'One': 1, 'two': 2}
```

and their values may be accessed like this:

```
>>> x = {'One': 1, 'two': 2, 'THREE': ['A', 'list', 'of', 'strings']}
>>> x['One']
1
>>> x['THREE']
['A', 'list', 'of', 'strings']
```

As you can see there are no restrictions regarding the type of the value that each key is pointing to. Keys on the other hand have to be hashable. In practice the keys are usually `ints` or `strings`. If you want to retrieve something that does not exist an error is raised:

```
>>> x = {'One': 1, 'two': 2, 'THREE': ['A', 'list', 'of', 'strings']}
>>> x['four']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
```

A way of providing a fallback in case a key does not exist in the dictionary is the `dict.get()` method:

```
>>> x = {'One': 1, 'two': 2, 'THREE': ['A', 'list', 'of', 'strings']}
>>> x.get('two', 'fallback')
2
>>> x.get('four', 'fallback')
'fallback'
```

## 5.5 Common Operations

### 5.5.1 Length

Sometimes you are interested in the amount of objects that a container holds. This information can be accessed using the `len()` function:

```
>>> x = [1, 2, 3]
>>> len(x)
3
>>> x = (1, 2)
>>> len(x)
2
>>> x = set([1, 2, 2, 3, 3, 'three'])
>>> len(x)
4
>>> x = {'One': 1, 'two': 2, 'THREE': ['A', 'list', 'of', 'strings']}
>>> len(x)
3
```

### 5.5.2 Membership Check

```
>>> x = [1, 2, 3]
>>> 2 in x
True
```

**Note:** While the behavior is clear for `lists`, `tuples` and `sets`, the `in` checks for the keys of a `dictionary`.

```
>>> x = {
...     'one': 'two',
...     3: 4
... }
>>> 'one' in x
True
>>> 'two' in x
False
>>> 3 not in x
False
>>> 4 not in x
True
```

Due to the nature of the implementation member checks are especially fast in `sets` and `dictionaries`

## 5.6 Summary

- use `lists` if the order is important and you may need to modify the container.
- use `tuples` if the order is important and the container is fixed.
- use `sets` if order is not important and you want to ensure uniqueness inside the container.
- use `dictionaries` if you want to store key-value pairs.

CHAPTER 6

# Flow Control

> "Would you tell me, please, which way I ought to go from here?"
> "That depends a good deal on where you want to get to," said the Cat.
> "I don't much care where—" said Alice.
> "Then it doesn't matter which way you go," said the Cat.
> "—so long as I get *somewhere*," Alice added as an explanation.
> "Oh, you're sure to do that," said the Cat, "if you only walk long enough."
>
> —Alice's Adventures in Wonderland

Up until now the code war rather... underwhelming. You only executed a sequence of commands each and every time. But with the introduction of the control flow of a program a whole new world of programs reveals itself. It is up to your experience and creativity to use the control flow tools to structure your code in a way that complex tasks can be handled.

## 6.1 Boolean Expressions

Knowing boolean expressions is essential for the control flow of a program. Python handles this rather intuitively. The most important boolean operations are

**==** equal to

**!=** unequal to

**<** less than

**<=** less than or equal to

**>** greater than

**>=** greater than or equal to

```
>>> 1 == 1
True
>>> 1 + 1 != 2
False
>>> 5*2 >= 11
False
```

**Warning:** Beware of `float` comparisons, as the way computers represent floating-point numbers internally might lead to some seemingly weird behavior.

```
>>> 0.1 + 0.2 == 0.3
False
```

This is certainly unexpected behavior. To deal with this shortcoming version 3.5 of Python introduced the `isclose` function in its `math` module.

```
>>> import math
>>> math.isclose(0.1 + 0.2, 0.3)
True
```

If you are working with an older version there are still convenience solutions. The `numpy` package that is introduced later on in this tutorial also provides the means for `float` comparisons.

In many cases a simple boolean expression is not sufficient to correctly specify the control flow of a program. Sometimes a code block should only be executed when a condition is not met, when several conditions are met all at once or when at least one of several conditions is met. For this Python hast the keywords `not`, `and`, and `or`, respectively. Their behavior is outlined in the following tables.

Table 6.1: Truth table for `not`

| A | not A |
|---|---|
| True | False |
| False | True |

Table 6.2: Truth table for `and`

| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

Table 6.3: Truth table for `or`

| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

Combining boolean expressions using these keywords is an essential skill for programming.

```
>>> i = 3
>>> i < 4 and i >= 0
True
>>> i < 4 and i > 3
False
>>> i < 4 and not i > 3
True
>>> i >= 0 or i > 3
True
```

You can furthermore use brackets to specify the order of the evaluation of the subexpressions like you would in equations.

**Note:** In Python `False` is not the only thing that is "false" in Python. `False`, `None`, numeric zero of all types, and empty strings and containers are all interpreted as false.

The most important construct using boolean expressions is introduced in the following.

## 6.2 The `if` Statement

A construct every programming language provides is the if statement. The basic structure is as follows:

```
if <expression>:
    <code block>
```

The result is that the *code block* is only executed when *expression* is `True`.

**Note:** Test

```
>>> x = 0
>>> y = 1
>>> if x > 0:
...     print('Larger than 0')
...
>>> if y > 0:
...     print('Larger than 0')
...
Larger than 0
```

The *expression* can also include a negation using Boolean operations:

```
>>> x = 0
>>> y = 1
>>> if not x > 0:
...     print('Not larger than 0')
...
Not larger than 0
>>> if not y > 0:
...     print('Not larger than 0')
...
```

If you want to cover both cases you can also use the else keyword:

```
>>> x = 1
>>> if x < 0:
...     print('Negative')
... else:
...     print('Positive')
...
Positive
```

But as you can see this does not cover all the cases. What if x is 0? For this we have to use elif:

```
>>> x = 0
>>> if x < 0:
...     print('Negative')
... elif x == 0:
...     print('Zero')
... else:
...     print('Positive')
...
Zero
```

And you can add as many elifs as you want.

## 6.3 The `while` Loop

Sometimes it is necessary to perform a routine until a certain condition is met. This is achieved using a while loop.

```
>>> x = 0
>>> while x < 5:
...     print(x)
...     x += 1
...
0
1
2
3
4
```

Assume you want to exit the while loop when a certain condition is met. This is possible with the break statement.

```
>>> x = 0
>>> while x < 5:
...     if x == 3:
...         break
...     print(x)
...     x += 1
...
0
1
2
```

**Attention:** Although while loops are a common building stone in every programming language I advise you to avoid using them whenever possible. It happens quite easily that the criterion for exiting the loop is never reached and your program gets stuck performing the same task more often than you intended. In many cases a while loop can be substituted with a for loop.

## 6.4 The `for` loop

In a lot of cases you just want to work on all the elements of a container one at a time. This is easily achieved with for loops.

```
>>> x = [1, 2, 3]
>>> for i in x:
...     print(i)
...
1
2
3
```

Here `i` takes on the value of the elements of `x` one after the other. This allows you to work with `i` inside of this for loop. After all elements have been visited you automatically exit the loop. A more sophisticated example might be to store the squared values of another list in a new list.

```
>>> x = [1, 2, 3]
>>> x_squared = []
>>> for value in x:
...     x_squared.append(value**2)
...
>>> print(x_squared)
[1, 4, 9]
```

### 6.4.1 `range`

A shortcut to loop over integers is given as the `range()` function.

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for i in range(3, 6):
...     print(i)
...
3
4
5
>>> for i in range(3, 12, 3):
...     print(i)
...
3
6
9
```

### 6.4.2 `enumerate`

Sometimes you also want to track where you currently are in your iteration. For example you want to know what the current state of your program is, but printing the value you are operating on each single time is kind of too much. Then you could use `enumerate()` like this:

```
>>> results = []
>>> for i, value in enumerate(range(100,900)):
...     if i % 200 == 0:
...         print('Current step:', i, '-- Value:', value)
...     results.append(i**2 % 19)
```

```
...
Current step: 0 -- Value: 100
Current step: 200 -- Value: 300
Current step: 400 -- Value: 500
Current step: 600 -- Value: 700
```

As you can see we now have comma-separated variables `i` and `value`. `i` get the current index we are in whereas `value` holds the actual object of the container.

### 6.4.3 zip

Another common task is that you have to loop over several lists at the same time. Use the :func:`zip` function for this:

```
>>> fruits = ['banana', 'orange', 'cherry']
>>> colors = ['yellow', 'orange', 'red']
>>> for fruit, color in zip(fruits, colors):
...     print('The color of', fruit, 'is', color)
...
The color of banana is yellow
The color of orange is orange
The color of cherry is red
```

**Note:** `zip()` stops the for loop as soon as one `list` is empty:

```
>>> fruits = ['banana', 'orange', 'cherry', 'apple', 'lemon']
>>> colors = ['yellow', 'orange', 'red']
>>> for fruit, color in zip(fruits, colors):
...     print('The color of', fruit, 'is', color)
...
The color of banana is yellow
The color of orange is orange
The color of cherry is red
```

## 6.5 Errors

As Python is a dynamic language it can never be garantueed that the input of your functions is what you want it to be. Take as an example a function whose purpose is the computation of the sum of the digits in a number. What if by accident someone passes a string as argument to this function? In some cases it is hence a good idea to check the input of a function for its sanity. If the input does not hold this test you may raise and error like this:

```
>>> raise ValueError('The input was wrong')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: The input was wrong
```

## 6.6 Summary

- if statements are a way to add a branch into your code.

- while loops should be avoided whenever possible.
- for loops are used to work on items of a container.
- `range()` is used to ad-hoc generate containers holding integers.
- `enumerate()` is used to keep track of the current index in a for loop.
- `zip()` is used to loop over several containers at once.
- raise is used to raise an error when something is fishy.

CHAPTER 7

# Functions

## 7.1 Introduction

Some tasks in your code may be performed a lot of times. Say you want to say "Hello" not only to the world but to a lot of people. You would have to write

```
print('Hello, {name}!')
```

an awful lot of times. It would be way shorter if we were able to write

```
greet('{name}')
```

and it would just print `Hello,{name}!`. OK, it is not that shorter... But imagine you want to find the roots of some functions. There is way more code involved and having a nice little shortcut to execute all that code without writing it over and over again would be nice, wouldn't it? This is in the spirit of the DRY principle! As a nice side effect you only have to change the routine in one place of your code instead of all of them if you find a bug or want to use some more sophisticated routine.

But even if you do only use a code snippet once in your program it may be more declarative to give it a short name and hide the implementation somewhere else. This also helps you to divide and conquer larger problems!

The nice thing is that you already know a function: `print()`! You used it like this:

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

Let's take a closer look at the parts that make up a function.

## 7.2 Components of a Function

### 7.2.1 Name

Most functions have names [1]. The name of `print()` is – never would have guessed – *print*. Printing the function without calling it reveals that it is a built-in function:

```
>>> print(print)
<built-in function print>
```

There are a lot more Built-in Functions, among them, for example, `float()`.

[1] Lambdas are the exception to the rule as they define anonymous functions.

### 7.2.2 Positional Arguments

`float()` is a function that lets you convert a number or a string that represents a number to a floating point number. Its interface is defined as

**float** (*x*)

where `x` is its argument. To be more precise it is its *positional argument*. By calling `float()` and passing an integer as argument the result is the corresponding float:

```
>>> float(1)
1.0
>>> float(-2)
-2.0
```

You can also pass strings that represent numbers to `float()`:

```
>>> float('1')
1.0
>>> float('-2')
-2.0
>>> float('1.500')
1.5
>>> float('1e-2')
0.01
>>> float('+1E6')
1000000.0
```

A similar function also exists for integer. It is the function `int()`.

### 7.2.3 Keyword Arguments

`int()` is a function that lets you convert a number or a string that represents a number to an integer. Its interface is defined as

**int** (*x*, *base=10*)

where `x` is its *positional argument* and `base` is its *keyword argument*. While positional arguments must be passed to a function keyword arguments are optional and provide default values. The default value of `base` is `10`.

When passing floats to `int()` everything after the decimal point is dropped:

```
>>> int(1.0)
1
>>> int(-2.0)
-2
>>> int(1.3)
1
>>> int(1.8)
1
>>> int(-1.3)
-1
>>> int(-1.8)
-1
```

When passing strings to `int()` the `base` keyword argument is used to indicate how the number should be interpreted in terms of which base it is given in.

```
>>> int('1')
1
>>> int('-2')
-2
>>> int('101010')
101010
>>> int('101010', base=2)
42
```

You do not have to write out the keyword arguments, they will be interpreted in the same order as they are in the interface:

```
>>> int('101010', 2)
42
```

But you can not convert strings that represent floating point numbers to integers like this:

```
>>> int('1.0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0'
```

For this you would have to chain `int()` and `float()`:

```
>>> int(float('1.0'))
1
```

## 7.3 Defining a Function

A function is defined by starting with the `def` keyword. Subsequently you provide the name of the function and its arguments in paranthesis. You can then use it like the functions you already used.

```
>>> def greet(name):
...     print('Hello', name)
...
>>> greet('World')
Hello World
>>> greet('you')
Hello you
```

```
>>> def add_reciprocal(a, b):
...     return 1/a + 1/b
>>> add_reciprocal(4, 8)
0.375
```

To provide keyword arguments you let the argument have a default value by assigning a value to it:

```
>>> def name_and_favorite_food(name, favorite_food='pizza'):
...     return name + "'s favorite food is " + favorite_food + '.'
...
>>> name_and_favorite_food('Dominik')
"Dominik's favorite food is pizza."
>>> name_and_favorite_food('Stefan', 'kimchi')
"Stefan's favorite food is kimchi."
```

If you execute them in the interactive Python interpreter the value returned by a function is automatically displayed if you do not assign it to a variable. It is important to know that the `return` keyword is used to make the value accessible to the outer scope, that is outside of the function. See the following example, where we first do not have a return statement:

```
>>> def is_positive_without_return(x):
...     if x >= 0:
...         print(True)
...     else:
...         print(False)
...
>>> is_positive_without_return(1)
True
>>> is_positive_without_return(-1)
False
>>> a = is_positive_without_return(1)
True
>>> print(a)
None
```

As you can see the variable `a` has no value at all. If you want `a` to receive the result of the function you have to return it, not print it:

```
>>> def is_positive_without_return(x):
...     if x >= 0:
...         return True
...     else:
...         return False
...
>>> is_positive_without_return(1)
True
>>> is_positive_without_return(-1)
False
>>> a = is_positive_without_return(1)
>>> print(a)
True
```

## 7.4 Functions as Function Arguments

You are able to pass almost anything to a function—even functions itself! This is especially useful if you want to do data fitting or root finding of mathematical functions.

```
def format_heading(text):
    return '\n' + text + '\n' + '='*len(text) + '\n'

def prettify(sections, header_formatter):
    # Assume that `sections` is a list of dictionaries with the keys
    # ``heading`` for the heading text and ``content`` for the content
    # of the section.
    # Assume that the `header_formatter` is a function that takes a string
    # as argument and formats it in a way that is befitting for a heading.
    text = ''
    for section in sections:
        heading = section['heading']
        content = section['content']
        text += format_heading(heading) + content + '\n'
```

```
    # Before returning it we make shure that all surrounding whitespace is
    # gone.
    return text.strip()

secs = [
    {
        'heading': 'Introduction',
        'content': 'In this section we introduce some smart method to teach Python.'
    },
    {
        'heading': 'Results',
        'content': 'More than 42 % of participants in this study learned Python.'
    }
]

pretty_text = prettify(secs, header_formatter=format_heading)
print(pretty_text)
```

So as you can see the argument `header_formatter` is just treated as if it was a function. If you run this script the output will be:

```
Introduction
============
In this section we introduce some smart method to teach Python.

Results
=======
More than 42 % of participants in this study learned Python.
```

## 7.5 Summary

- Functions can make your life easier by streamlining repeated tasks or giving a name to complex programming logic.
- Functions may have two different kinds of arguments, *positional arguments* that must be given to the function and *keyword arguments* which provide default values.
- Functions are defined using the `def` keyword.
- Functions may return values by using the `return` keyword.

## 7.6 Exercises

### 7.6.1 Heaviside Step Function

The Heaviside step function can be defined as

$$H(x) = \begin{cases} 0, & x < 0, \\ 1, & x0. \end{cases} \tag{7.1}$$

Take the following template, complete it, and test it:

```
def heaviside_step_function(x):
```

### 7.6.2 Newton's Method (1D)

Newton's method is a rather popular iterative root finding algorithm. Starting at an initial guess $x_0$ it tries to find better and better approximations of the root of a function $f(x)$. For this it uses the first derivative $f'(x)$ of the function. The process

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

is repeated until a value $f(x_n)$ is reached that is within a predefined tolerance to zero. For further information see the Wikipedia page.

The way it is supposed to work is as follows:

```
>>> def f(x):
...     return x**2 - 2
...
>>> def df_dx(x):
...     return 2*x
...
>>> newtons_method_1d(f, df_dx, x0=4, tol=1e-8)
1.4142135623730951
```

CHAPTER 8

# Strings

You have worked with them for a while now: `strings`. They offer a large amount of ways to work with text, of which a few selected ones are outlined in the following.

## 8.1 split

Working with a whole string at once is needlessly complicated in many cases. Hence the Python standard library offers a way to take a `string` apart: `str.split()`

```
>>> some_string = 'My first string'
>>> some_string.split()
['My', 'first', 'string']
```

By default it splits to an arbitrary amount of whitespace

```
>>> some_string = 'My     second     string'
>>> some_string.split()
['My', 'second', 'string']
```

**Note:** In Python whitespace is everything you can not directly see like spaces, tabs (`\t` in strings), and newlines (`\n` in strings).

but you can also specify another string which is used to split the other string apart:

```
>>> some_string = '0.1, 0.2, 0.3, 0.4, 0.5'
>>> some_string.split(',')
['0.1', ' 0.2', ' 0.3', ' 0.4', ' 0.5']
```

The string that is used to split with is consumed in the process. So to get rid of the additional whitespace you might get the idea to add it to the string used for the splitting

```
>>> some_string = '0.1, 0.2, 0.3, 0.4, 0.5'
>>> some_string.split(', ')
['0.1', '0.2', '0.3', '0.4', '0.5']
```

Eventually this might lead to problems if the string format is not strictly kept:

```
>>> some_string = '0.1,   0.2,0.3, 0.4,  0.5'
>>> some_string.split(', ')
['0.1', '  0.2,0.3', '0.4', ' 0.5']
```

But Python's got you covered...

## 8.2 strip

To get rid of unwanted whitespace around a string you can use the `str.strip()` method:

```
>>> some_string = '\n\n    So much surrounding whitespace\n\n'
>>> print(some_string)


    So much surrounding whitespace


>>> some_string.strip()
'So much surrounding whitespace'
```

You can also get rid of something else than whitespace by specifying the characters as an argument:

```
>>> more_string = '...Python...'
>>> more_string.strip('.')
'Python'
```

---

**Note:** The characters you specify as an argument are not seen as a string but as a collection of characters you want to strip of the string:

```
>>> incantation = 'abracadabra'
>>> incantation.strip('bra')
'cad'
```

---

## 8.3 Summary

- Use `str.split()` to split a `string` at a substring.
- Use `str.strip()` to get rid of excess characters at the edges, especially whitespace.

## 8.4 Exercises

### 8.4.1 Split and Strip

From time to time you may want to split a string by a delimiting character like `,`, but the whitespace is all over the place.

```
>>> some_string = '0.1,   0.2,0.3, 0.4,  0.5'
>>> some_string.split(',')
['0.1', '   0.2', '0.3', ' 0.4', '  0.5']
>>> some_string.split(', ')
['0.1', '  0.2,0.3', '0.4', ' 0.5']
```

So a pure splitting operation does not really do the trick. Write a function that improves this behavior for this case, so that you can use it as follows:

```
>>> strip_and_split(some_string, ',')
['0.1', '0.2', '0.3', '0.4', ' 0.5']
```

You can take the following template:

```
def split_and_strip(string, delimiter):
```

CHAPTER 9

# File IO

Usually the result of your program is important not only now but in the future as well. For this it is a good practice to store the results in file so that you can work with them later on if necessary. This is called file input and output, or in short: file IO.

## 9.1 Reading from a File

The best way to open a file in Python is by using the with statement. This automatically opens the file, keeps a lock on it so that no one is able to modify it while you are working with it, and closes it afterwards. Interacting with the file is then done by working with the variable that you specify after `as`. This is called a file object. To get the complete content of a file use its `read()` method like this:

```
>>> with open('csv_file.txt', 'r') as f:
...     file_content = f.read()
...
>>> print(file_content)
1, 2, 3
4, 5, 6
7, 8, 9
```

where the first parameter is the path to the file and the second parameter is the so-called file mode. `r` is for read-only.

**Note:** In some tutorials and also production code you may find something along the lines of this to interact with a file:

```
>>> f = open('csv_file.txt', 'r')
>>> file_content = f.read()
>>> f.close()
>>> print(file_content)
1, 2, 3
4, 5, 6
7, 8, 9
```

This is kind of the old style of working with files when the with statement did not exist yet. It has the huge downside that you have to take care about closing the file. And if an error is raised between `open` and `close` it is not closed at all. The with statement takes care of this for you even in the case of an exception and is subsequently the preferred way.

You can also iterate over `f` as if it is some form of container:

```
>>> with open('csv_file.txt', 'r') as f:
...     for i, line in enumerate(f):
...         print('Line', i, '--', line)
...
Line 0 -- 1, 2, 3

Line 1 -- 4, 5, 6

Line 2 -- 7, 8, 9
```

As you can see you get some additional white lines. The reason for this is that each line still contains its newline character `\n` in the end. To this one `print()` adds an additional one by default so you end up with an empty line. To circumvent this use the `strip()` method of the line string:

```
>>> with open('csv_file.txt', 'r') as f:
...     for i, line in enumerate(f):
...         stripped_line = line.strip()
...         print('Line', i, '--', stripped_line)
...
Line 0 -- 1, 2, 3
Line 1 -- 4, 5, 6
Line 2 -- 7, 8, 9
```

## 9.2 Writing to a File

To write to a file you have to open it first, this time with `w` as file opening mode, to indicate that you want to write to the file. Then you can use the `write()` method of the file object to write to the file:

```
with open('my_first_file.txt', 'w') as f:
    f.write('This is smart.')
    f.write('This is even smarter.')
```

Now the content of your file would be

```
This is smart.This is even smarter.
```

Which is not nicely formatted. So you have to take care that you add the newline character `\n` and spaces accordingly.

```
with open('my_first_file.txt', 'w') as f:
    f.write('This is smart.\n')
    f.write('This is even smarter.\n')
```

## 9.3 Summary

- You can open a file using the `open()` function in a `with` statement.
- To merely read from a file open it with the filemode `r` and use the `read()` method of the file object.
- To write from a file open it with the filemode `w` and use the `write` method of the file object.

CHAPTER 10

# NumPy

NumPy is a third-party package for Python that provides utilities for numerical programming with Python. It is commonly imported via

```
import numpy as np
```

which maps the NumPy package to the name `np`. So every time you see some code with the `np` prefix you may assume that NumPy is imported even though it might not be stated explicitly.

CHAPTER 11

# Array

The NumPy `array` is the underlying mechanism that makes NumPy so convenient and efficient.

## 11.1 Creating Arrays

A NumPy `array` is easily initialized via

```
>>> np.array([0, 1, 2])  # 1D array of integers
array([0, 1, 2])
>>> np.array([0.0, 1.0, 2.0])  # 1D array of floats
array([ 0.,  1.,  2.])
>>> np.array([[0, 1], [2, 3]])  # 2D array of integers
array([[0, 1],
       [2, 3]])
```

So by nesting lists of numbers you are able to construct multi-dimensional arrays. But there are also other methods to initialize `arrays`:

- `numpy.zeros()`:

```
>>> np.zeros(3)
array([ 0.,  0.,  0.])
>>> np.zeros([2, 3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

- `numpy.zeros()`:

```
>>> np.ones(3)
array([ 1.,  1.,  1.])
>>> 3 * np.ones(3)
array([ 3.,  3.,  3.])
>>> np.ones([2, 3])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

- `numpy.eye()`:

```
>>> np.eye(2)
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> -2 * np.eye(2)
```

```
array([[-2., -0.],
       [-0., -2.]])
```

- `numpy.arange()`: This one should feel rather familiar to the `range` of regular Python. But where the latter is only able to deal with integers, the implementation of NumPy can also work with floats.

```
>>> np.arange(5)
array([0, 1, 2, 3, 4])
>>> np.arange(3, 7)
array([3, 4, 5, 6])
>>> np.arange(2, 4, 0.5)
array([ 2. ,  2.5,  3. ,  3.5])
```

- `numpy.linspace()`: If you need evenly spaced samples this way of initializing them is preferred to `numpy.arange()` due to the latter introducing slight roundoff errors which is caused by the implementation.

```
>>> np.linspace(0, 4, 5)
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.linspace(0.3, 0.7, 5)
array([ 0.3,  0.4,  0.5,  0.6,  0.7])
>>> np.linspace(0.3, 0.7, 4, endpoint=False)
array([ 0.3,  0.4,  0.5,  0.6])
```

## 11.2 Array Attributes

The `arrays` also provide some information about themselves which can be accessed by its attributes.

### 11.2.1 Number of Dimensions

The `ndim` attribute of an array is the amount of dimensions of the array.

```
>>> x = np.array([0.0, 1.0, 2.0])
>>> x.ndim
1
>>> x = np.array([[0, 1, 2], [3, 4, 5]])
>>> x.ndim
2
```

### 11.2.2 Shape

The `shape` attribute of an array is a tuple representing the number of elements along each dimension.

```
>>> x = np.array([0.0, 1.0, 2.0])
>>> x.shape
(3,)
>>> x = np.array([[0, 1, 2], [3, 4, 5]])
>>> x.shape
(2, 3)
>>> x.shape[0]
2
```

```
>>> x.shape[1]
3
```

### 11.2.3 Size

The `size` attribute of an array is the amount of elements of the array.

```
>>> x = np.array([0.0, 1.0, 2.0])
>>> x.size
3
>>> x = np.array([[0, 1, 2], [3, 4, 5]])
>>> x.size
6
```

So essentially it is the product sum of the `shape`.

## 11.3 Accessing Data

Similarly to `lists` and `tuples` data is accessed via referring to the indices:

```
>>> x = np.array([[0, 1, 2],
...               [3, 4, 5]])
>>> x[0, 0]
0
>>> x[0, 1]
1
>>> x[1, 0]
3
>>> x[1, 2]
5
```

## 11.4 Slicing

Slicing refers to extracting partial data from arrays. This is very efficient as nothing is copied in memory.

```
>>> x = np.array([[0, 1, 2, 3, 4],
...               [5, 6, 7, 8, 9],
...               [10, 11, 12, 13, 14],
...               [15, 16, 17, 18, 19],
...               [20, 21, 22, 23, 24]])
>>> x[0, :]
array([0, 1, 2, 3, 4])
>>> x[1, :]
array([5, 6, 7, 8, 9])
>>> x[:, 1]
array([ 1,  6, 11, 16, 21])
>>> x[:3, :3]
array([[ 0,  1,  2],
       [ 5,  6,  7],
       [10, 11, 12]])
>>> x[2:, 2:]
```

```
array([[12, 13, 14],
       [17, 18, 19],
       [22, 23, 24]])
```

**Note:** Slices of an array share the memory of the original array. Hence all the changes you do to a slice are also represented in the original array:

```
>>> x = np.array([[0, 1, 2, 3, 4],
...               [5, 6, 7, 8, 9],
...               [10, 11, 12, 13, 14],
...               [15, 16, 17, 18, 19],
...               [20, 21, 22, 23, 24]])
>>> print(x)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> x_slice = x[1:-1, 1:-1]
>>> print(x_slice)
[[ 6  7  8]
 [11 12 13]
 [16 17 18]]
>>> x_slice[1, 1] = 888
>>> print(x_slice)
[[  6   7   8]
 [ 11 888  13]
 [ 16  17  18]]
>>> print(x)
[[  0   1   2   3   4]
 [  5   6   7   8   9]
 [ 10  11 888  13  14]
 [ 15  16  17  18  19]
 [ 20  21  22  23  24]]
```

CHAPTER 12

# Math

The goal of NumPy is to provide functions and classes that make numerical computations easier. For an extensive overview see the section Linear algebra (numpy.linalg). of the NumPy documentation. Some of them require the part `linalg` after `numpy`. In combination with

```
import numpy as np
```

the functions and classes may be used via

```
np.linalg.<function>
```

The functions that require this import are indicated by this usage.

## 12.1 Componentwise Math Operations

NumPy `arrays` support the typical math operations introduced in *Basics* and they are executed componentwise. Here are a few examples:

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> y = np.array([6, 5, 8, 9, 10])
>>> z = 2.0
>>> x + z
array([ 3.,  4.,  5.,  6.,  7.])
>>> x + y
array([ 7,  7, 11, 13, 15])
>>> y % z
array([ 0.,  1.,  0.,  1.,  0.])
>>> y / x
array([ 6.        ,  2.5       ,  2.66666667,  2.25      ,  2.        ])
```

## 12.2 Dot

If you need a scalar product for 1D `arrays` or matrix multiplication for 2D `arrays` the function `numpy.dot()` is the function of choice:

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> y = np.array([6, 5, 8, 9, 10])
>>> np.dot(x, y)
```

```
126
>>> x.dot(y)  # alternative syntax
126
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]])
>>> y = np.array([[10, 11, 12],
...               [13, 14, 15],
...               [16, 17, 18]])
>>> z = np.array([[1/19, 1/20, 1/21],
...               [1/22, 1/23, 1/24],
...               [1/25, 1/26, 1/27]])
>>> np.dot(x, np.dot(y, z))  # this is a bit convoluted
array([[ 12.35196172,  11.80535117,  11.30555556],
       [ 29.63712919,  28.32591973,  27.12698413],
       [ 46.92229665,  44.84648829,  42.9484127 ]])
>>> x.dot(y).dot(z)  # better
array([[ 12.35196172,  11.80535117,  11.30555556],
       [ 29.63712919,  28.32591973,  27.12698413],
       [ 46.92229665,  44.84648829,  42.9484127 ]])
```

## 12.3 Eigenvalues

The function `numpy.linalg.eig()` may be used to compute the eigenvalues and eigenvectors of a square array:

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]])
>>> np.linalg.eig(x)
(array([  1.61168440e+01,  -1.11684397e+00,  -1.30367773e-15]), array([[-0.23197069, -
↪0.78583024,  0.40824829],
       [-0.52532209, -0.08675134, -0.81649658],
       [-0.8186735 ,  0.61232756,  0.40824829]]))
```

The output is a bit messy, but ss you can read in the documentation of `numpy.linalg.eig()` the function actually returns two things: an array holding the eigenvalues and an array holding the eigenvectors as columns:

```
>>> eigenvalues, eigenvectors = np.linalg.eig(x)
>>> for i in range(eigenvalues.size):
...     print(eigenvalues[i], eigenvectors[:, i])
...
16.1168439698 [-0.23197069 -0.52532209 -0.8186735 ]
-1.11684396981 [-0.78583024 -0.08675134  0.61232756]
-1.30367772647e-15 [ 0.40824829 -0.81649658  0.40824829]
```

## 12.4 Norm

Computing the norm of a vector is a fairly common task, so it seems obvious that NumPy also provides a function for this: `numpy.linalg.norm()`.

```
>>> x = np.array([3, 4])
>>> np.linalg.norm(x)
5.0
```

Besided the regularly used $\ell^2$-norm this function also offers other norms. For further information use the documentation of `numpy.linalg.norm()`.

## 12.5 Determinant

Knowing the determinant of an array may tell you whether it is singular or, in the case of a $3x3$ array, tell you the volume of the rhomboid that is spanned by the vectors composing the array.

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 4],
...               [3, 2, 1]])
>>> np.linalg.det(x)
-7.9999999999999982
```

This should be $-8$ so it is an interesting way to see that the determinant computed by NumPy is computed numerically and not analytically.

## 12.6 Exercises

### 12.6.1 Gaussian Elimination (Eye Variant)

Solving systems of linear equations is one of the basic tasks in numerical mathematics – hence it is also one of the basic tasks in computational materials science. A system of linear equations like

$$\begin{gathered} 2x + y - z = 8 \\ L_1 \\ -2x - y + 2z = -11 \\ L_2 \\ -2x + y + 2z = -3 \\ L_3 \end{gathered}$$

may also be described via

$$\vec{A}\vec{x} = \vec{b}$$

where

$$A = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} \quad x = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad b = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}$$

The solution algorithm makes use of the augmented matrix form

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$$

The procedure is then to first get this augmented matrix form into triangle form and subsequently form the identity matrix on the left hand side. The right hand side then is the solution. For further information see Gaussian elimination

Start with the following template, complete it, and test it:

```
def gaussian_elimination_eye(A, b):
```

CHAPTER 13

# Matplotlib

> "The purpose of visualization is insight, not pictures."
>
> —Ben Shneiderman

Visualization is one of the most important means for the interpretation of the results of scientific computations. The package that is mostly used for this task is Matplotlib.

The `matplotlib` package is typically imported via

```
import matplotlib.pyplot as plt
```

And subsequently all plotting commands are called using the prefix `plt`.

Rather than exercises that want you to get some results you are shown simple examples for the most common types of plots.

CHAPTER 14

# Line Plots

The most common plot you see in scientific publications may well be a so-called line plot. The command for line plots is `plot()` and using it like this

```
import matplotlib.pyplot as plt
import numpy as np

# Generate data for the plot
x = np.linspace(0, 1, 21)
y0 = np.sin(2*np.pi*x)
y1 = np.cos(2*np.pi*x)

# Generate the plot
plt.plot(x, y0)
plt.plot(x, y1)
plt.show()
```

results in a line plot with both the sine and cosine:

CHAPTER 15

# Scatter Plots

When your data does not have a sequence but is still characterized by data points a so-called scatter plot is the plot of choice. The command is `scatter()` and when used like this

```
import matplotlib.pyplot as plt
import numpy as np

r = np.random.RandomState(42)
x = r.random_sample(10)
y0 = r.random_sample(10)
y1 = r.random_sample(10)

plt.scatter(x, y0, color='blue')
plt.scatter(x, y1, color='green')
plt.show()
```

it results in a plot showing the different data point groups:

CHAPTER 16

# Pcolormesh Plots

pcolormesh()

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.ndimage.filters import gaussian_filter

x = np.linspace(0, 1, 51)
y = np.linspace(0, 1, 51)
r = np.random.RandomState(42)
z = gaussian_filter(r.random_sample([50, 50]), sigma=5, mode='wrap')
z -= np.min(z)
z /= np.max(z)

plt.pcolormesh(x, y, z)
plt.colorbar()
plt.show()
```

CHAPTER 17

# Contour Plots

contour()

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.ndimage.filters import gaussian_filter

x = np.linspace(0, 1, 50)
y = np.linspace(0, 1, 50)
r = np.random.RandomState(42)
z = gaussian_filter(r.random_sample([50, 50]), sigma=5, mode='wrap')
z -= np.min(z)
z /= np.max(z)

plt.contour(x, y, z)
plt.colorbar()
plt.show()
```

CHAPTER 18

# Contourf Plots

contourf()

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.ndimage.filters import gaussian_filter

x = np.linspace(0, 1, 50)
y = np.linspace(0, 1, 50)
r = np.random.RandomState(42)
z = gaussian_filter(r.random_sample([50, 50]), sigma=5, mode='wrap')
z -= np.min(z)
z /= np.max(z)

plt.contourf(x, y, z)
plt.colorbar()
plt.show()
```

CHAPTER 19

# SciPy

SciPy is a collection of modules that offer high-level interfaces to algorithms that are common in the scientific community, like linear algebra, optimization, integration, interpolation, etc. Covering all the possibilities in a tutorial is almost impossible, so to see what SciPy can do refer to its documentation.

CHAPTER 20

# Fitting Curves

The routine used for fitting curves is part of the `scipy.optimize` module and is called `scipy.optimize.curve_fit()`. So first said module has to be imported.

```
>>> import scipy.optimize
```

The function that you want to fit to your data has to be defined with the `x` values as first argument and all parameters as subsequent arguments.

```
>>> def parabola(x, a, b, c):
...     return a*x**2 + b*x + c
...
```

For test purposes the data is generated using said function with known parameters.

```
>>> params = [-0.1, 0.5, 1.2]
>>> x = np.linspace(-5, 5, 31)
>>> y = parabola(x, params[0], params[1], params[2])
>>> plt.plot(x, y, label='analytical')
>>> plt.legend(loc='lower right')
```

Then some small deviations are introduced as fitting a function to data generated by itself is kind of boring.

```
>>> r = np.random.RandomState(42)
>>> y_with_errors = y + r.uniform(-1, 1, y.size)
>>> plt.plot(x, y_with_errors, label='sample')
>>> plt.legend(loc='lower right')
```

Now the fitting routine can be called.

```
>>> fit_params, pcov = scipy.optimize.curve_fit(parabola, x, y_with_errors)
```

It returns two results, the parameters that resulted from the fit as well as the covariance matrix which may be used to compute some form of quality scale for the fit. The actual data for the fit may be compared to the real parameters:

```
>>> for param, fit_param in zip(params, fit_params):
...     print(param, fit_param)
...
-0.1 -0.0906682795944
0.5 0.472361903203
1.2 1.00514576223
>>> y_fit = parabola(x, *fit_params)
```

```
>>> plt.plot(x, y_fit, label='fit')
>>> plt.legend(loc='lower right')
```

As you can see the fit is rather off for the third parameter `c`. A look at the covariance matrix also shows this:

```
>>> print(pcov)
[[  1.64209005e-04   1.75357845e-12  -1.45963560e-03]
 [  1.75357845e-12   1.16405938e-03  -1.73642112e-11]
 [ -1.45963560e-03  -1.73642112e-11   2.33217333e-02]]
```

But to get a more meaningful scale for the quality of the fit the documentation recommends the following:

```
>>> print(np.sqrt(np.diag(pcov)))
[ 0.01281441  0.03411831  0.15271455]
```

So the fit for `a` is quite well whereas the quality `c` is the worst of all the parameters.

---

**Note:** This routine works by iteratively varying the parameters and checking whether the fit got better or worse. To help the routine find the best fit it is hence a good idea to give it a good starting point. this can be done using the `p0` argument of `curve_fit()`. In some cases this is even necessary.

---