Principles of Programming and Computational Foundations

Martin Werner

Nov 08, 2022

CONTENTS

Download as PDF

This repository is an Open Educational Resource (OER) capturing various topics on computation both for students from aerospace and geodesy. Of course, all content appearing on this page in the future is going to be something we believe being of high importance for **both** geodesy and aerospace. But maybe not at the same time during study or maybe not for every focus area inside these sciences. Hence, students and self-learners are asked to always critically review which parts are of most value for them.

In order to clarify the aspects really relevant to the lecture audiences and especially to possible examinations, we prepare two tables of contents for both lectures.

Principles of Programming (2022)

Principles of Programming is a one-semester master course building on profound knowledge of object-oriented design, programming in Java, and geospatial data in all its forms by adding aspects of software engineering, the high-performance language C++ (in its modern form) and Python.

Extended ToC - Principles of Programming

Computational Foundations (2022)

Computational Foundations is a two-semester course in the aerospace bachelor and aims at laying out fundamentals of working with computers including aspects from theoretical computer science, computer engineering, and programming selected for their relevance in aerospace engineering.

Extended ToC - Computational Foundations

Part I

Lecture and Tutorial Information

CHAPTER

ONE

PRINCIPLES OF PROGRAMMING - EXTENDED TABLE OF CONTENTS

1.1 Material

- Lecture 1: Basics / States / Operating Systems / Command Line
  - *The Disk Operating System (DOS)*
  - *Microsoft Windows*
  - *Unix and Linux*
- Lecture 2: Imperative Programming
  - *Imperative Programming*
  - *Python*

Preview Material:

- The Notion of an Algorithm

1.2 Background and Context

Principles of Programming is an advanced (master) course in the study program Geodesy and Geoinformation. This course assumes very good knowledge of at least one object-oriented programming language (typically Java), object-oriented design and data modeling (UML, etc.), as well as geospatial data and basic geodata algortihms.

Based on this knowledge (which is typically taught in the Bachelor degree program at Technical University of Munich, the aim of this lecture is described as

> With this course, students are enabled to solve their geodetic problems using programming. They obtain a working knowledge of programming languages, algorithms, software patterns and libraries needed to solve complex computatoinal problems.

in the module definition and we give the following framework of aspects that we have to cover:

> In this course, students are introduced to advanced programming and algorithms with examples in Python and C++. It shall as well include some examples from computational geometry, point cloud processing, image analysis, etc. to - at the same time - introduce canonical patterns or even core libraries for working with data from subfields of geodesy.

- Python, numpy, tensorflow
- C++11, generic programming (beyond object orientation which I expect them to “roughly” know)

- Python & C++ interaction
- Various Data Structures and techniques

1.3 Time Plan

This lecture is organized into thirteen units out of which you can influence quite a few depending on the previous knowledge in the group and the interest.

First the dates in 2022/23:

Date	Topic
18.10.2022	Lecture + Tutorial
25.10.2022	Lecture + Tutorial
01.11.2022	Public Holiday
08.11.2022	Lecture + Tutorial
15.11.2022	Conflict (in resolution)
22.11.2022	Lecture + Tutorial
29.11.2022	Lecture + Tutorial
06.12.2022	Lecture + Tutorial
13.12.2022	Lecture + Tutorial
20.12.2022	Lecture + Tutorial
	Christmas Break
10.01.2023	Lecture + Tutorial
17.01.2023	Lecture + Tutorial
24.01.2023	Lecture + Tutorial
31.01.2023	Lecture + Tutorial
07.02.2023	Lecture + Tutorial
———	———————

1.4 Modules

tba

CHAPTER

TWO

COMPUTATIONAL FOUNDATIONS - EXTENDED TABLE OF CONTENTS

2.1 Computational Foundations I

2.1.1 Material

- Lecture 1: Basics / States / Operating Systems / Command Line
  - *The Disk Operating System (DOS)*
  - *Microsoft Windows*
  - *Unix and Linux*
- Lecture 2: Imperative Programming
  - *Imperative Programming*
  - *C++ - First Steps*

Preview Material:

- The Notion of an Algorithm

2.1.2 Time Plan

This lecture is organized into thirteen units out of which you can influence quite a few depending on the previous knowledge in the group and the interest.

First the dates in 2022/23:

Date	Topic
18.10.2022	Lecture
25.10.2022	Lecture
01.11.2022	Public Holiday
08.11.2022	Lecture
15.11.2022	Conflict (in resolution)
22.11.2022	Lecture
29.11.2022	Lecture
06.12.2022	Lecture
13.12.2022	Lecture
20.12.2022	Lecture
	Christmas Break
10.01.2023	Lecture
17.01.2023	Lecture
24.01.2023	Lecture
31.01.2023	Lecture
07.02.2023	Lecture
———	———————

2.1.3 Modules

tba tba

2.2 Computational Foundations II

tba

Part II

Basic Knowledge

CHAPTER

THREE

THE DISK OPERATING SYSTEM (DOS)

In order to fully understand Microsoft Windows and in order to get to advanced usage capabilities, it is unavoidable to understand, how Microsoft Windows has emerged. In the old days of computing, a company named Microsoft introduced an operating system known as Disk Operating System (DOS) which was used to run most personal computers in these days. From a user perspective, this operating system did all the work of starting up the computer and configuring hardware and running programs. Therefore, a command line was designed and equipped with programs and technology to support basic computing tasks.

MS Dos was a single tasking operating system (except TSRs, which have been small programs that were able to run in the background). This dictates the logic under which interaction with MS Dos is cut into sequential steps. In a nutshell, the computer tells the user that it waits for an instruction by showing a command prompt.

```
C:\>
```

The user is then supposed to enter a command which then runs a program or a builtin instruction. Only four types of information are available to both the user and programs:

- The current working directory (CWD) which is a combination of a drive and a folder relative to which all file operations are performed
- A set of environment variables, such as the %PATH% variable
- The name of the program and a space-separated list of arguments to the command

In this context, the DOS provided a set of commands for working with these states and software vendors could provide additional programs. While DOS can be considered history, all current Windows versions include a Command Prompt feature, which provides a DOS-like command line to perform tasks on Windows computers. For an overview, we list a few DOS commands and ask you to explore them yourself on a Windows computer (or in FreeDOS in a virtual machine like Virtual Box).

Each command starts with the name of the command or program and a set of arguments where arguments starting with a "/" are considered switches that just influence the behavior. Many file-oriented commands allow you to use wildcards. A wildcard in a string matches zero or more character (*) or exactly one unspecified character (?). For more clarity:

- stat*.bat would match status.bat as well as stat.bat
- data?.dat would match data0.dat, but not data10.dat

Basic programs

- <drive letter>: to change the drive
- CD to change the directory
- DIR to list a directory, consider switches /P and /S which change the behavior
- MD to create a directory
- RD to remove a directory (only if empty)

- TREE shows all files below the current working directory
- ATTRIB show and modify attributes like write protection on files
- COPY is used to copy files
- DEL deletes files (synonymous with ERASE)
- EDIT provides a simple editor (EDLIN before MS DOS 6.0)
- FIND searches for a string in a file
- MORE pages a file to the screen
- MOVE moves a file to a different location
- PRINT is used to print a file
- REPLACE works like copy but replaces the file in the target
- TYPE outputs the whole content of the given file
- XCOPY extends COPY to be able to copy whole directories and trees
- CLS clears the screen
- DATE shows and modifies the date
- TIME shows and modifies the time
- ECHO is used to control whether commands are shown or not (mainly in batch files)
- FDISK is used to set up hard disks (partitions, etc.)
- FORMAT organizes a file system on floppy disks or hard disk partitions
- HELP shows help for a dos command (use it in the tutorial!)
- SET shows configuration information and environment variables and modifies them
- VER shows the version of DOS in use

With these commands, it is possible to organize a computer quite nicely yet remaining simple and self-explaining. Another interesting aspect about DOS which is visible in modern versions of Windows is the fact that file names were heavily restricted in early versions allowing 8 characters for the file name and 3 characters for the file name extension separated with a dot. Extensions have always been used to mark the type of file, for example article.txt was a text file suitable for the edit program while word.exe was a program known as Word which could be run by writing the command word (without the extension) when in the same directory or when the directory was mentioned in the PATH variable.

Note: Assignment 1:

- Install VirtualBox and run FreeDOS inside
- Using the FreeDOS command line, create a folder structure representing Germanys federal structure. Start by creating a folder Germany, within this folder, create one for each state.
- In each state, create a file capital.txt and write into it only the name of the capital of the state in one line (be sure to end the line)
- Show the tree (and submit as a solution) using the command line
- (Advanced) Output all member states
- Learn about the redirection of output using the > pipe

- Look around on your Windows PC if you have one. Where is your data stored technically, where is Documents located? Where are Downloads? Try to find out about this by just running the "Command Prompt" or "Eingabeaufforderung" in German language.
- Create a file on the Desktop of your Windows Computer, use the extension .txt and write Hello Windows into it. Then open the file with the Windows GUI which should spill up your favourite editor.

DOS used drive letters A, B, C, etc. to distinguish different disks. In the very early days, one typically had no hard drive in an MS DOS computer, but one floppy disk. This disk was known as A and all the life was taking place in A:>. A bit later, many computers came up with a second floppy drive. Now, the drive A was used to start the operating system (DOS) and provide commands, while B:> was used to store data. As floppy disks are sometimes usable (when a valid disk has been inserted) and sometimes unusable, both letters are reserved up to today. Typically, the first drive letter assigned to hard drives or other modern devices is, therefore, C:>. In almost all versions of Windows today, Windows is installed on a drive with letter C. If the main drive has more partitions, drive letters are used sequentially, such that a two-partition setup often has a drive D:>. On many other computers, D: already refers to some CD drive or USB stick. In a nutshell, drive letters are assigned along the alphabet and as the first two letters are reserved for floppy drives, the first letter in every-day use is C. As a consequence of this unknown dynamics, it has become a tradition to map the first network drive with the letter Z and to continue backwards (if you are in a network-enabled environment). Furthermore, some companies have started to use a drive called H like "Home" for the home drive of a user.

Note: Assignment Two: Install Windows (at least once)

Windows Installation Tutorial. When you start working with Windows a lot, you will face the situation that your computer is not working properly anymore. In this tutorial (accessible only for people with a valid Windows license), we will install Windows into a virtual machine to train the procedure. In order to help you for your future, we ask you to do this on a virtual disk of 20 GB (use a FCOW disk to save space) which you partition into three disks: Disk C (the main Windows disk) shall use 10 GB, while a drive D of 5 GB and a drive E of 5 GB shall be available as well. Therefore, you can use the partitioning tool part of the Windows installer. We will use Windows Education 11, but the procedure is not much different for any (unmodified) Windows.

CHAPTER

FOUR

AUTOMATION IN WINDOWS

Since the beginning (including MS DOS), the builtin functionality can be used to automate routine tasks to some extent. In Windows, batch files are being used. In their simplest form, batch files just list a sequence of commands to be executed one after another. But batch files can have more aspects such as looping over files, asking for user input and other advanced patterns. As many of our students will be exposed to a Windows with no access to advanced scripting software, it is worth knowing some basic aspects of how to write batch files on Windows.

As said, a batch file is just a text file in ASCII format and each row of the file represents a command which you could as well type into the command prompt. However, it gets more interesting if one realizes the following three functionalities:

- Disabling the output
- Looping over files
- Using arguments given to the batch file

The first aspect is simple, but important: in basic batch files, each command is first output before the command is executed and the output of the execution is output. This means that it is a bit tricky to have concise output from a batch script which is really useful for the user. In a batch file, the output of a command can be suppressed by prefixing it with @ why it is good practice to start all batch files with a line

```
\@echo off
```

The output of this line is suppressed (it would typically output "ECHO is disabled") and the output of commands is disabled for the remainder of the script.

Now, it is very common that batch files are used to automate annoying commands to avoid typing and thereby avoiding typos as well. Imagine you are working on satellite images and you have downloaded 1,000 scenes from Sentinel 2. As you want to create a web map, you decided to reproject all of these files from their own projection into a WebMercator projection (EPSG 90009001). You figured out the right parameters for letting gdal_warp utility doing your work, for example

```
gdal\_warp \<scene\> -t\_srs ... (please figure it out yourself!)
```

Then, you can use a batch file like this one:

```
\@echo off
for %%f in (\*.tif) do (
gdal\_warp \<magic parameters here\> %%\~nf output/%%\~nf
)
```

This file takes all files with extension tif in the current directory and gives them as input to our magic gdal_warp command while putting the result into the same file, but in a directory output. Hence, be sure to have this directory created (or make the creation of this directory part of the batch script itself). In this way, you can now have a lot of coffee or go to bed while your computer is working through your archive of data.

Note: Assignment Three: Writing Batch Files

Create a batch file which reproduces the result of Assigment One. A batch file in Microsoft DOS and Windows is a simple ASCII text file with an extension of .BAT. Those files can be run from the command prompt just like EXE files by giving their name without extension. Use the @ECHO OFF as the first line to suppress the output of the individual programs. Explore on your own using the Internet, how one can get input from the user into a variable, how one can loop over files running a certain command for each file.

Note: Assignment Four: Learn some Latin

We will write a batch file program to train latin vocabulary. Therefore, we create a file with the latin name containing the translation (followed by a newline). Then, try to write a script that first selects a file at random (or any other way) and asks (by showing the filename representing the latin vocabulary). Then ask the user to input and (!maybe!) try to compare the user input with the file contents at least roughly. Note that this can be really tricky in BATCH programming, so any approximate solution is appreciated.

*Just some tips: To make it less complex, the solution to this task was not comparing the results, but rather showing the user input and the translation from the file. Further, we create two Batch files: one which loops over all files and another one which is called with the filename and presents a short dialog. *

Note: Assignment Five: Reproject Sentinel Scenes

Download a handful of Sentinel scenes from different locations, install GDAL and use the gdal_warp command line utility to bring all the files into the same projection.

CHAPTER

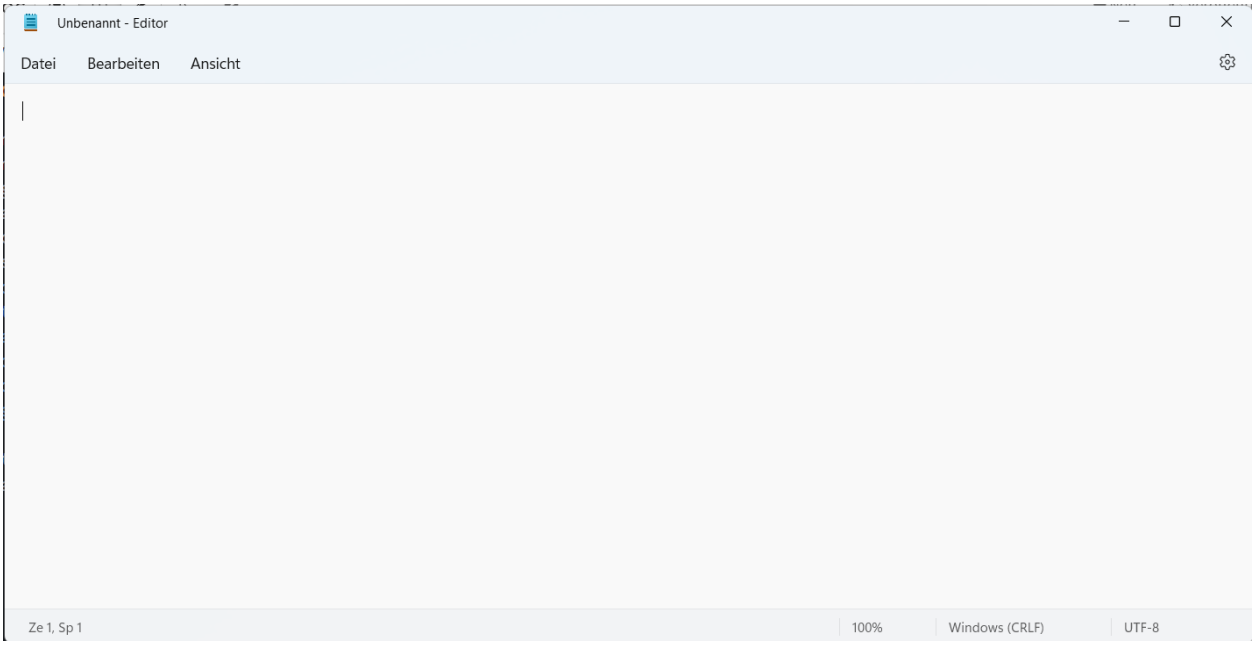
FIVE

MICROSOFT WINDOWS

The Windows operating system has then emerged as a graphical user interface (GUI) on top of DOS. In its early versions, it was a graphical file manager, but the most important innovation was available with Windows 3.x, namely, an interaction scheme in which the graphical screen is subdivided into rectangular Windows with the following properties

- Windows can overlap each other
- Exactly one Window is active
- The active Window is in the front (fully visible)
- Windows can be resized and moved on the screen
- Windows provide buttons for minimizing, maximizing and closing the Window

In fact, a Window today looks like this editor window from the integrated Windows Editor Notepad:



Windows typically have more standardized aspects such as the following ones. Each Window has a **Title** line which contains an optional Icon (which is a menu) followed by a text (the Window title) and the three buttons on the right for minimizing, maximizing and closing the Window. Below the Title, a **menu bar** is located in which multiple text fields are displayed. When you click on them, a window is opened and can be navigated to send a command to the application. The bottom line (optional) is known as a status line and typically contains a few controls, at least one text. The remainder is called the Client Area and is used by the application.

In order to give you a more in-depth understanding, let us look at the principle of writing programs for MS Windows. A normal program is started, this one asks the Operating System to open a window with certain properties (size, location, etc.). One of these properties is a function which is then called by the operating system with events such as the following ones:

- WM_CREATE: Is sent when the Window is created
- WM_DESTROY: The window is destroyed
- WM_MOVE: The location has changed
- WM_SIZE: The size has changed
- WM_ACTIVATE: The window has become active
- WM_QUIT: The X has been clicked (or an equivalent hotkey was activated)
- WM_PAINT: Draw the client area (however this is done)

Without expecting everyone to understand completely at the first time you read this article, here is a complete Windows API example drawing a rectangle. It is written in the C language we will learn, and serves as a primary example. It has been made available at Github and is referenced by the MSDN as well MSDN Article on WM_PAINT

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int nCmdShow)
{

    // Register the window class.
    const wchar_t CLASS_NAME[]  = L"Sample Window Class";

    WNDCLASS wc = { };

    wc.lpfnWndProc   = WindowProc;
    wc.hInstance     = hInstance;
    wc.lpszClassName = CLASS_NAME;

    RegisterClass(&wc);

    // Create the window.

    HWND hwnd = CreateWindowEx(
        0,                              // Optional window styles.
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style

        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
        NULL        // Additional application data
```

(continues on next page)

(continued from previous page)

```
        );

    if (hwnd == NULL)
    {
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);

    // Run the message loop.
    MSG msg = { };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // All painting occurs here, between BeginPaint and EndPaint.
            FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
            EndPaint(hwnd, &ps);
        }
        return 0;
    }

    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

This very clearly illustrates what is happening: The program creates a new Window which refers to a function in our program (`WindowProc`) and this function is called by ourselves with all messages we can Peek and Dispatch in our main program.

The nature of an event-driven system is that after a lot of initialization, the main progress of the system follows an event-driven nature: an ordered sequence of information (called events) will dictate the behaviour.

By the way, this is also the core reason why sometimes windows are hanging and not reacting: This happens, when messages like WM_PAINT or WM_CLOSE are not delivered, because the event loop is not running properly. Windows typically blurs out the Window and shows a dialog about this problem.

As a consequence, good programs will have to make sure that all messages are quickly handled maybe by making other parts of the program proceed asynchronously, for example in a thread.

It is interesting to see how a concurrent impression has been created by using the graphical user interface, which is inherently non-parallel: there is only one active program at a time.

CHAPTER

SIX

UNIX, LINUX, AND POSIX

The Linux family of operating systems has been improving in importance in the last decades. The project started as an illustrative implementation of a simple UNIX kernel on top of the 80386 computer and is now the leading operating system in terms of global impact.

In order to understand Linux, one has to look back into the history as well. Linux is modeled after Unix which is a family of operating systems developed for mainframes. Such computers have been very expensive and available long before the personal computer was available, but a single UNIX mainframe was used by many members of a company.

Hence, topics like user management, access control and parallel execution of different things (for different users) have been at the heart of Unix development, while the simplicity (despite all risks) of the DOS immediate mode was not accessible.

At this point, before looking into the guiding principles of Unix and Linux, we can mention a name here: the basic Unix was mainly developed by Ken Thompson and Dennis Ritchie who has also had huge impact on the development and success of the C programming language. In fact, Unix has been implemented mainly in assembler (a rather raw machine language with almost no abstractions), but later translated in large parts into the C language. And this is one of the earliest predecessors of the programming language we will focus on.

Furthermore, a few principles have been fixed in the early days and have proven successful enough not to be changed that much in the years to come. This group of decisions is referred to as the Unix philosophy, which we will revisit when talking about software and programming.

A famous formulation of the Unix philosophy is due to Douglas McIllroy:

- Write programs such that they do only one thing and they do it well
- Write programs such that they can work together
- Write programs such that they work on textual streams as this is the universal interface

This is often oversimplified to

> Do only a single thing at a time and do it well

Or (more or less the same) as the KISS principle:

> Keep it simple stupid.

There is a lot to these aspects and we will learn a lot about it. More down to the point, Mike Gancarz gives the following list

- Small is beautiful.
- Make each program do one thing well.
- Build a prototype as soon as possible.
- Choose portability over efficiency.
- Store data in flat text files.

- Use software leverage to your advantage.
- Use shell scripts to increase leverage and portability.
- Avoid captive user interfaces.
- Make every program a filter.

This will guide our gentle introduction to Linux in the sequel.

6.1 POSIX

Unix has led to a joint understanding of how computers should work and it has been very successful. But it has also quickly become an area of debate (see Unix Wars). In order to unify and converge the market, a new standard has been carefully designed and developed (ISO/IEC/IEEE 9945) under the name POSIX.

This standar describes clearly and independent from a concrete implementation defintions of terms, the system interface (concretely in C language including header files), and the command line interpreter and a list of tools.

When learning C (or any other modern language), one will often touch exactly this standard even in a non-Unixoid environment (like Windows).

We will now skip over the definitions and the header files, as they naturally appear when learning programming, but jump right ahead to the list of mandatory utilities that you can expect any unix-like operating system to provide:

For a complete list, please refer to https://pubs.opengroup.org/onlinepubs/9699919799/idx/utilities.html

For our first interaction with these systems, we selected

- ar (nowadays often tar for tape archiver)
- at
- awk
- basename
- bc
- cat
- cd
- chgrp
- chmod
- chown
- compress (nowadays, bz2 and gzip)
- cp
- cut
- date
- dd
- df
- diff
- dirname
- du

- echo
- ed
- env
- ex
- expand
- expr
- false
- fg
- file
- find
- fold
- getopts
- grep
- head
- iconv
- id
- jobs
- join
- kill
- ln
- ls
- man
- mkdir
- mkfifo
- more
- mv
- nl
- paste
- patch
- printf
- ps
- pwd
- read
- rm
- rmdir
- sed

- sh (nowadays often bash)
- sleep
- sort
- split
- tail
- tee
- test
- time
- touch
- tr
- true
- ulimit
- umask
- uname
- uniq
- unlink
- vi (nowadays vim)
- wait
- wc
- who
- xargs
- zcat (gzcat, bzcat)

Note: Install a Linux, preferably Debian (with no tasks selected, video follows), and login to the system with the user account you created during installation. Then, inform yourself about the commands using the `man` command.

6.2 Linux as seen by a user

In order to understand Linux, we need to understand the command line version of it. With Linux being a multiuser operating system, our adventure starts with logging in to Linux by giving a username and its associated password. Then, we end up with a command line, most typically running the Bourne Again Shell (bash).

Similar to the DOS command line, the system now waits for your instructions and gives you some state information like the current working directory (PWD on Linux, accessible with the `pwd`comamnd). Again, you can now use the programs given to start working with a Linux computer.

First, we will have to navigate the system and in Linux there is no concept of drive letters. Instead, there is one file system root (/) and from there the journey begins.

We can move around by using cd, we can as well use the special directories . to refer to the current directory and .. to refer to the parent directory in the relevant contexts.

Note: Navigate to the main directory, then from there into the bin directory. This directory traditinoally holds user programs. By using a pipe character, you can make the output of a program becoming the input of another such that `ls | less` will let you page through all programs in the installation.

Navigate to /home. Show all directories using ls. This should now have one directory per user. Enter whoami to find out your user name.

Navigate to /etc, where Linux configuration is held. All programs should keep their configuration there as a plain text file. Look at the file /etc/fstab using an editor of your choice (vi, vim, emacs, nano).

Navigate to /proc. This holds kernel information and files to interact with the kernel. Look at /proc/meminfo, /proc/partitions and /proc/meminfo to find some information about your computer.

6.3 Example: Processing data in a Linux environment

As long as you stick with the Unix principles, Linux provides sufficient tools for 99% of your everyday tasks as a data scientist. No advanced software or programming is required in this context.

In order to illustrate the line of thinking, let us perform a rather simple task: count the words in the definition (RFC) of the HTTP protocol available from RFC

```
curl https://www.rfc-editor.org/rfc/rfc2616.txt > 2616.txt
```

will just download the document. To simplify the remainder, we create a local copy, so be sure to be in a reasonable directory (maybe create one) first.

In order to find the most frequent word, our task is now to break it down into individual words. We do this by relying on the `tr` tool. Read the man page, but we just turn every space into a newline:

```
cat rfc2616.txt | tr " " "\n"
```

In this command, the file is first output to stdout, but this standard output is bound to the standard input of the tr command. This takes every space (first argument) and turns it into a newline. The output is ugly and long (would keep scrolling for quite some time), hence, we can rely on head to see part of it

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | head -10




Network
Working
Group

martin@martin:~/lecture$
```

Okay, this looks nice, but empty lines will be dominating (this happens, because there are a lot of spaces in the document for layouting page numbers). Let us get rid of empty lines as follows (this is tricky, we will discuss this in the tutorial)

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
 ↪head -10
Network
Working
Group
R.
Fielding
Request
for
Comments:
2616
UC
```

Now, we can start counting the words. The easiest way to do this is to sort the output using `sort` and then use the `uniq -c` command to remove successive equal lines outputting the count of removed lines. Let us again limit the amount of screen usage with head:

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
 ↪sort |uniq -c | head -10
      1 "
      1 ""
      1 ""%"
      1 "#"
      1 "%
      3 "("
      2 ")"
      1 ")">
     26 "*"
      3 "*",
```

Here, now the first column shows the number of times a string has been seen and we need to find the largest ones. We do this by sorting again, but numerically using `sort -g` and reversing the direction. As we might be unsure whether it works, we can again work on reduced outputs by keeping the head in the command.

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
 ↪sort |uniq -c | head -10 | sort -rg
     26 "*"
      3 "*",
      3 "("
      2 ")"
      1 ")">
      1 "%
      1 "#"
      1 ""%"
      1 ""
      1 "
```

Now, we are almost there: Let us now really look for the ten most frequent words in RFC 2616:

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
 ↪sort |uniq -c | sort -rg | head -10
   3532 the
   1579 a
   1349 to
   1298 of
```

(continues on next page)

(continued from previous page)

```
   1006 is
    829 and
    773 in
    661 that
    653 be
    550 for
martin@martin:~/lecture$
```

When you take your time to learn a few (if not all) of the core utilities (GNU coreutils), you can solve almost all problems based on text files considerably faster than with any other environment.

CHAPTER

SEVEN

IMPERATIVE PROGRAMMING

Programming is the principles procedure of telling a computer what it is supposed to do and there are various programming styles that can be used to program computers. However, the most widely used and most basic style of thinking of computers is the paradigm of imperative programming.

Imperative programming matches very well the nature of computers being rather dumb and simple machines. In imperative programming, the programmer takes the role of an imperator and provides instructions to the computer in a very precise ordered form. These instructions come from a comparably small set of instructions which the computer must support.

7.1 A Robot Model of Intrinsic Instructions

A very basic model of a computer can be imagined as a small robot that has two functions: turn right by an angle of $\pi/2$ and go one step into the direction currently looking at. When building such a machine, one immediately also designs a programming language which has two operations (we call them intrinsics, because they are the operations really implemented in hardware). Let us give names to them: `move` for moving one step, and `turn` for turning left by $\pi/2$.

Now an imperative program is very much like a shell script: from the language that we have defined, we can create a text file with one intrinsic instruction per line (this form of machine code is typically called assembler).

The program

```
move
turn
turn
move
```

would thus be interpreted as an imperative program like: first move into the direction you are facing, then turn to the left, then turn again to the left, then move again.

So far, we have introduced a few concepts that should be highlighted:

- intrinsic instructions are those steps that a physical computing machine can perform immediately
- Each intrinsic instruction gets a name
- A series of intrinsic instructions can be written into a file which is an imperative program and interpreted as a sequential instruction

Let us reflect a bit more on the situation of such a robot. In a certain sense, the previous program does define very precisely what the robot is supposed to do, but neither in which state (location and orientation) it has been in the beginning nor in which state it is afterwards. In fact, each operation is well-defined local to the robot (we know how the wheels are to be moving), but not well-defined with respect to the robot in the world. Therefore, we would need to give or fix initial conditions.

For a real imperative program, these intiial conditions are the state that is held in the operating system about the program, for example, the current working directory (CWD).

7.2 Procedures

A very typic additional definition in an imperative programming setting is the notion of a procedure. A procedure is an imperative program (e.g., a sequence of instructions) such that this sequence can be referred to as a new operation in the programming language. These are the so-called non-intrinsic operations.

For our robot, the following program

```
turn
turn
turn
```

means to turn left for three times. This, of course mimicks the result of turning right. In many programming languages, there is a way to make this a new instruction of the language called a procedure.

```
proc turnright:
  turn
  turn
  turn
```

Our minimal computer (the robot) with its tiny set of intrinsic instructions (move and rotate) can now be programmed with a third instruction `turnright`

7.3 Ports and Functions

As a next step in the co-evolution of a computer and a programming language, one might want to be able to react on real-world input, that is, something coming from outside the robot. For a robot, we could imagine a sensor that just tells us whether the place we would move to is occupied or not.

That is, we extend our hardware with something we will call a port as it brings external information into the system. And we will extend the programming language with an intrinsic function to model this port.

A function is a procedure in the sense that it can be run and that it can be built together from other instructions and functions, but it **returns a value.** For the case of the port letting us experience the next location being occupied, this return value can have two states: occupied or not occupied. Due to the huge impact of George Bool on the behavior of such values (this British mathematician passed away already in 1864 long before digital computers have been realized), any two-valued information in a computer is called a Boolean value or bool for short.

Concretely speaking, a function would now look like `is_empty` and when this is `called` it would turn into the current value associated with the hardware port.

Now, with having ports, we can observe aspects of the surroundings (information exterior to the system itself) and in order to react to them, two aspects are introduced in imperative programming:

- control structures and
- expressions

7.4 Control Structures

As we are still looking for a minimal programming language, we could imagine that our robot is supposed not to crash with the surroudings, so maybe we just need to be able to make instructions like move **conditional** to the value of the port.

This control structure is often nown as if .. then .. else .. More concretely, we enable the following snippet of source code:

```
if is_empty then move else turn
```

This program would now always first check the condition of the if (run the intrinsic function) and if move is possible, it would move and if not, it would turn in the hope that we can move afterwards.

To complete this exposition of minimal control structures, there is another common way to use boolean information in imperative programs: to control the repeated execution of something. To this end, we introduce a loop called while:

```
while is_empty do move done
```

This means: as long as it is possible to move, continue moving. Note that by introducing a Boolean function, we would typically also introduce the two Boolean values as constants for our programs. They are typically referred to as true and false, hence, we can also write

```
while true do
  while is_empty do
    move
  done
  turn
done
```

7.5 Scopes

Another concept we silently introduced in the previous example is the idea of scopes. It is so common that within a conditional branch or a loop multiple instructions need to be placed that we try to avoid to introduce a function for it. Because if we had to, we would make things into functions that are used only once. As an alternative, a **scope** is introduced which is a sequence of instructions that is taking the role of a single instruction like the loop body (what to do as long as a condition is true) or the two branches of a conditional information (what to do when the value is true, what to do otherwise).

Scopes have varying notations: sometimes with braces (C++, Java)

```
{
move
turn
}
```

sometimes with indentation (Python)

```
while true:
  move
  turn
```

sometimes with barrier words (Bash)

```
while true do
move
turn
done
```

sometimes with round brackets (DOS/Windows BAT files)

```
(
)
```

But they all serve the same purpose of quickly and locally (on the screen in the right location) bundling together instructions.

CHAPTER

EIGHT

MEMORY

Now, this very small robot can be extended further to interact with a physical world. As you may have noticed, the robot is currently limited to a grid of points it can reach as we only implement movement by $\pi/2$. Assume now, we give the robot some small things it can deposit into the grid cell he is currently in, and then also sense, and maybe even pick up. That is, we extend the robot device with three functions represented by three intrinsics:

- a function deposit to put something into the current cell. Let us assume that it fails if the cell is already filled with such an item.
- a function pick to clean the current cell
- a function has_item to check if the current cell is filled with an item.

With these three intrinsics, we can write a lot of algorithms and it can be fun. The interesting aspect is that the 2D world provides us with the ability to have a concept like a variable.

8.1 Variables

A variable is an area of memory to hold information together with an interpretation associated with this information. For the 2D robot case, a variable is often such a thing as a stack of items like in the following drawing. The identity of the variable being the column in the grid while the value is the height of the stack.

In all imperative programming languages, the idea is similar: there is an axis of varying identity (think like rows in a table) and each variable typically has a name (just a string) in order to use it. The second axis in programming languages describes together the amount of space that is needed and the interpretation of the space. For example, you can have a 32 bit integer number (reserving 32 bits = 4 bytes of memory) assigned an identity like `i` or the same amount of memory for a 32 bit floating point number `f`.

In high-level languages, it is not uncommon that variables are accessible by name during runtime, in lower level languages, the names are only available during compilation and will be removed (for efficiency) while compiling the source code to a executable.

8.2 An example robot program sketch

Assume we have our robot being on the lowest block of a tower of blocks each with a marker. Let us assume, we want to interpret the height of this tower as an integer number, say `h` like height. Let us assume further, we want to compute the value `2h`. How would we proceed?

8.2.1 Algorithm Design

It is pretty clear and intuitive, how the robot can solve the problem: For each marked block, it creates two marked blocks somewhere else. More concretely, let me lay down a proposal as follows:

- The robot starts on the lowest block of a tower
- The robot walks upwards to the highest block
- The robot picks up the marker (the tower has reduced in height by one)
- The robot goes down as long as markers are there (to the ground floor, so to say)
- The robot goes a step to the right
- THe robot walks up the tower as long as possible going to the first non-marked place
- The robot deposits a marker
- The robot goes up
- The robot deposits a marker
- The robot goes up
- The robot deposits a marker
- The robot goes down as long as possible
- The robot goes left (and is back in the state we want)

This needs to be repeated until we have taken the whole first tower.

Now, with this overview of the algorithm, we continue with an implementation strategy: there are quite a few things that are semantically bounded in the sense that we can precisely and simply describe the state before and after a part of our program. We could for example implement a procedure `climb` and use it twice: once to climb up the left tower and once to climb up the second tower. Which tower to climb will just be based on the current location of the robot.

Our program skeleton grows slowly. In order to have a more narrative programming style, we introduce another idea of programming, namely to put human readable text comments into the source code to illustrate some aspects and further to use such comments to specify the expectations of implementations. Note that some modern programming languages like C++ allow us to have such specifications during compilation in one or another way, but this is a rather new feature and has not yet been voted into many standards. But the topic to keep an eye on is known as contracts.

```
# function climb
# precondition: the robot looks up
# invariant: the robot keep the same X coordinate
# postcondition: has_marker == false && has_marker @ below would be true
def climb:
  # needs to be written
```

Only with this contract information, it is possible to write a semantically correct climb function, especially the precondition is requried: otherwise we would never know where we are going and the robot would need a compass or other means to find its orientation.

Let us complete this procedure at least:

```
# function climb
# precondition: the robot looks up
# invariant: the robot keep the same X coordinate
# postcondition: has_marker == false && has_marker @ below would be true
def climb:
```

(continues on next page)

(continued from previous page)

```
  while has_marker:
    climb
```

Hence, we can start writing a part of our program completely assuming we are in the start location (bottom of left tower):

```
climb
turn
turn
move
pickup
climb # this now climbs down
```

Now we realize that our first intuition that there is a climb function is not a very good and compact one as we also have to climb down. So let us update the precodnition

```
# function climb
# precondition: the robot looks up or down
```

We can then extend the program by going to the side

```
...
turn
move
turn
```

Now, we are one block to the side looking up again. We can then deposit markers twice:

```
climb  # go up
deposit # marker one
climb  # go up
deposit # marker two
turn    #look down
turn
climb   # walk down
turn   # look back
turn
turn
move
turn    # up in origin
```

This completes our imperative formulation with a function climb that we have used multiple times making it much easier to use. And we have seen that even our simple robot can do computations (doubling an integer number represented as the height of a stack of markers).

In fact, similar robots are used to teach programming. For example, Richard Pattis has introduced Karel the robot with a very similar set of features as our imaginary robot and Nikolaus Wirth has popularized this in Germany as Niki the robot

Actually, I found that Niki is still available in a historic Windows version and it runs at least on my Windows 11. So please go ahead and have a look at Niki the robot.

CHAPTER

NINE

THE C AND C++ LANGUAGE

9.1 Introduction

The C programming language is the earliest form of the C and C++ family of programming languages. Nowadays, plain C is used in kernel programming (e.g., the Linux kernel), in long-established code bases (e.g. postgresql), and in very resource-constrained situations (e.g., embedded devices). It also presents a subset of modern C++ and (with very few incompatibilities), every modern C++ compiler will compile plain C source code without any issues.

C and C++ are compiled languages and the compilation process is decomposed into a few tasks and translations that can run independetly from each other. A modern GUI will orchestrate these steps, but it is mandatory to understand the translation procedure to some extent when learning C++.

To keep things simple, a C project typically contains a bunch of C (extension *.c) or C++ (extension *.cpp) files. These contain source code which a *compiler* can translate to object code. Object code is a form of machine code which is not executable as it can be incomplete. In a second phase, all object codes are linked together with system libraries such that all missing symbols are either coming from one of the object files of the project (hence, from your source code) or from some libraries (either system or additional libraries).

In order to deal with the situation that your source code depends on functionality that is not available to the compiler, because it is in a different file or coming from the operating system libraries, the C system provides a second type of file in which only the *signature* of functions is defined. In this way, the compiler is able to generate code to understand the libraries contents (it is kind-of a table-of-contents) well enough to compile the object file without looking into the actual implementation.

Summarizing, we have C++ files (*.c,* .cpp) and header files (*.h,* .hpp) and we translate each soure code file (*.c,* .cpp) into an object file (*.o) in order link together all objects with libraries (*.o, *.lib) into an executable (*.exe on Windows, just a name on Linux).

With this theory, we are set to start learning the core language while already understanding a bit the files that we will create.

9.1.1 Hello World

The following program is the first program you should learn and understand. It just prints out “Hello World” to the screen. We will provide four information aspects in this script: the source code file (C++), a Makefile (which contains the compiler arguments, at the moment just compiling, the output of compiling (could become interesting later) and the output of running the program (note that the first line is the invocation for the case we are later giving arguments). In the HTML version of this book, they are alternative with each other (tabs), in print, they are a bit lengthy, but in this way everything remains complete.

Source

```c
#include<stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Makefile

```
all:
        g++ -Wall -o 01_oldhelloworld 01_oldhelloworld.cpp
run:
        ./01_oldhelloworld
```

Build Output

```
```

Run Output

```
```

The file (like any other C++ files) first lists a few of header files to include (here stdio.h, defined in the POSIX standards, see https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdio.h.html.

From this library, we are relying on the printf function to just output a string.

C is an imperative language and instructs a computer step by step, the entry point (where this processing starts) is a special function called main (WinMain for Windows programs). Similar to what we have seen for the shell examples, the n denotes a special character "new line" and makes sure that the program does not only output Hello World!, but also continues to the next line.

Note: There is a very traditional incompatibility between Windows and Linux with respect to new lines: The Linefeed (LF, n) is sufficient on Linux and Unix, but on Windows, lines are typically ending with CR-LF: first a carriage return (CR, r), then a line feed (n).

The printf function is a powerful utility you can find in all programming languages and it can format various values into the output, what we will cover a bit later, but it has also some downsides, mainly with respect to security and reliability. Hence, printf is not used that much in safety-oriented code.

9.1.2 Hello C++ World

In contrast to C, C++ has introduced the concept of an iostream which better reflects the nature of the standard input and standard ouput. A stream is an object that can represent a file, a network connection, or just a terminal window and one can stream various information into it. Therefore, an operator << is implemented, which takes variables, strings, or whatever information. In this way, type conversion is safe and automatic.

In order to output data to the user, the C++ libraries define a header iostreams which define cout (C output), cerr (standard error output) and cin (for input, used with a >> stream operator).

Hence, the same program written in idiomatic C++ looks like this:

Source

```
#include<iostream>

int main(void)
{
   std::cout << "Hello World!" << std::endl;
   return 0;
}
```

Makefile

```
all:
        g++ -Wall -o 00_helloworld.out 00_helloworld.cpp
run:
        ./00_helloworld.out
```

Build Output

```

```

Run Output

```

```

9.2 Functions and Signature-based Selection

A little bit later, we will introduce many different types, but for now, we will look into how we can write our own functions similar to the printf functions. A function is a unit of code that can be reused in your own code. Let us consider the following already quite involved example:

Source

```
#include<iostream>

int doubled(int a)
{
   return (a*2);
}

float doubled(float a)
{
   return (a*2);
}

int main(void)
{
   std::cout << "doubled(4)=" << doubled(4) << std::endl
             << "doubled(4.5)=" << doubled(4.5f) << std::endl;
   return 0;
}
```

Makefile

```
all:
        g++ -Wall -o 02_functionsignatures 02_functionsignatures.cpp
run:
        ./02_functionsignatures
```

Build Output

```

```

Run Output

```

```

While this example does not look that much more complicated, we introduced many aspects that are difficult to understand. Let us unroll it a bit. In C++, we are allowed to define multiple functions to have the same name. This is particularly wanted, because sometimes, we can implement an algorithm just different if the arguments are of different types. In our example, we define two times a double function, once it takes an integer argument (int, more on types later, it is just an integral number) or a fractional number (float).

The compiler selects the right function in a complex procedure: For the first call, `doubled(4)`, the type of the constant 4 is integer; hence, the system decides that the first function `int doubled(int)` should be called. For the second call, we have a problem: if we remove the f from the argument, the compiler will tell us the call is ambiguous. What the compiler actually means in this situation is that there is no function that strictly fits to the type (4.5 will be considered a double value in the first place, not a float value, more on this later). Hence, the compiler would generate code and covert the double into an integer if only the first function would exist, it would convert the value into a float if only the second variant would exist. But the compiler has no precedence and raises an error. We resolved this by using a literal: for many

constants, the type can be influenced by putting a character to the constant: 4.5f just means the value 4.5 as a floating point number of the type `float`.

9.3 Code Libraries

In many situations, we will implement functions that we want to use from multiple programs. For example, we could implement a sort function and use it in many different programs if we need to sort. In this situation, we need to put our implementations into an isolated C++ file, create a header file with the so-called *function prototypes* which are just fixing the name of the function and the type of the return value and of all arguments. In this way, we can include the header in our own program (one which has a main), generate code using the functions in the compilation step, and then link together our program with the library into a proper executable.

More concretely, we could move the two implementations of the `doubled` functions into their own C++ file:

Library

```
int doubled(int a)
{
   return (a*2);
}

float doubled(float a)
{
   return (a*2);
}
```

This one can now be compiled independently from everything else. Note that we do not even include iostream here, because we are not using anything from iostreams. In order to make this available, we formualte a header

Header

```
#ifndef DOUBLED_H_INC
#define DOUBLED_H_INC
int doubled(int a);
float doubled(float a);
#endif
```

which contains the signature lines of the functions only (ending with a semicolon). It is valid, but not very widely used to only give the types of the arguments such as

```
float doubled (float);
```

In order to make sure that these definitions cannot be loaded twice (not really needed here, but problems will be coming) all of this header is protected from double inclusion using the preprocessor. All lines with # are processed even before compiling and in order to avoid a double include (maybe you are including the file directly, but some library you include also includes this header) we check if a symbol specific to the header (it is common to use the capitalized filename as depicted) and if not, we define it (such that it is defined in the sequel) and close with `#endif` at end of file. In modern C++, this can be replaced with a simple `#pragma once` in the beginning, but this is not that widely used today.

The remaining project then looks like this:

Source

```
#include<iostream>
#include "03_doubled.h"

int main(void)
{
   std::cout << "doubled(4)=" << doubled(4) << std::endl
             << "doubled(4.5)=" << doubled(4.5f) << std::endl;

   return 0;
}
```

Makefile

```
all:
        g++ -c 03_doubled.cpp -o 03_doubled.o
        g++ -c 03_library.cpp -o 03_library.o
        g++ -o 03_library 03_library.o 03_doubled.o
run:
        ./03_library
```

Build Output

```

```

Run Output

```

```

A small thing to note here is that the include uses ” instead of < and >. This is an old relict and refers to being able to include from the current directory. Libraries to be included from the compiler default location are included with <> and things local to the project with “”.

Especially the Makefile (*.mk) is interesting here as I spelled out all three steps for you: compiling (-c) the library, then compiling the main program part. Afterwards linking together the object files into an executable.

Part III

Python (only Principles of Programming)

CHAPTER

TEN

PYTHON BASICS

(It has been taken from our previous educational project ICAML and will be reshaped to fit us better. **This page can be downloaded as** *interactive jupyter notebook*

This tutorial gives a brief overview of the most essential features of Python. I recommend to additionally read (at least) chapters 3, 4 and 5 from the official Python docs. Excellent Python tutorials are available at python-course.eu EN or python-kurs.eu GER. Another tutorial starting from zero is available at inventwithpython.com When searching the web for specific Python related topics, keep in mind that we are looking at **Python 3.X.X**. Python 2.7 is still very popular but shows some major differences in comparison with Python 3.

10.1 Table of contents

Basic interpreter rules Variables [*Data types* / *Naming variables*] *Operations* [*Arithmetic operations* / *Boolean operations* / *Comparisons*] *Conversions Decisions Built-in data types* [*Lists* / *Tuples* / *Dictionaries* / *Sets*] *Loops* [*While loop* / *For loop* / *Break and continue*] *Functions Built-in functions Classes Assertions Common errors Catching errors Lambda expressions Mixed topics*

1. Execute code statement-wise (top to bottom)

- Statements are usually separated by linebreaks

1. Execute statements by evaluating all expressions

- Start with the most nested expression
- Follow precedence rules

1. Ignore comments (#), blank lines and whitespace

Terminology:

- **Statements** can be seen as the instructions, any program consists of.
- A statement consist of at least one **expression**.

Note:

- Cells are executed by selecting them first and then pressing [CTRL]+[ENTER] (in the Jupyter Notebook)
- The output of a cell will be displayed below the cell.

```
a = 1          # 1 expression:  'assign the value 1 to variable a!'
b = a + 1      # 2 expressions: 'compute the sum of a and 1!' and 'assign result to␣
 ↪the variable b!'
c = a + b - 1  # 3 expressions: 'a + b' and 'X - 1' and c = X
print(c)       # 1 expression:  'call function print with argument c!'
print(c*10)    # 2 expressions: 'c*10' and print(X)
```

```
2
20
```

In Python the statements are usually separated by **line breaks**. Separating statements with a **semicolon** is also allowed but less common / readable. The following two cells are equal in terms of the statements but differently formatted. The arguments of print are texts (denoted by `'`):

```
print('Hello')  # first statement
print('World')  # second statement
# .. blank lines are ignored
print('!')      # third statement (.. white spaces between symbols / variables are␣
 ↪ignored as well)
```

```
Hello
World
!
```

```
# statements separated be semicolons -> valid but less readable
print('Hello'); print('World'); print('!')
```

```
Hello
World
!
```

Multi line statements are less common but also possible in Python. To continue a statement in the next line, we add `\` before the line break. Line continuation is also implied inside parentheses `( )`, brackets `[ ]` and braces `{ }`.

```
a = 1 + 2 + \
    3 + 4 + 5
print(a)

b = (1 + 2 +
     3 + 4 + 5)
print(b)
```

```
15
15
```

The most significant difference between Python and other languages is the **mandatory indentation** of blocks. Blocks are grouped statements (e.g. used in loops, conditions or functions). The following cell will demonstrate the indentation syntax. The `if`-keyword (details later!) introduces a block. Usually the indentation size is **four spaces**. Most IDEs (also the Jupyter server) will convert tabs to spaces so we can use tabs as well.

A simple indentation rule is: Whenever a statement ends with a colon, (at least) the next non-blank line has to be indented.

```
if True:   # 'if' is here just used to introduce a block. Statements that introduce␣
 ↪blocks end with a colon!
    print('Hello')
    print('World') # two statements inside a block (indented with four spaces / one␣
 ↪tab!)
print('!') # outside the block (no indentation)
```

```
Hello
World
!
```

Additional information about indentation and blocks is available here.

When creating a variable in Python, the data type is usually not given explicitly. Instead Python will automatically detect the variables type from the assigned value. To create and assign a variable the syntax is x = v, where x is the variables name and v the explicid value or an existing variable. When a variable already exists, it will be **overwritten** (and possibly also get a new data type). **Note** that variables in Jupyter Notebooks exist **across all cells.**

```
x = 1           # create x and assign the value 1
x = 3.5         # x already exists, so it will be overwritten with 3.5
y = x           # create y and assign the value of x
x = y = z = 1   # it is valid to assign a value to multiple variables
x,y,z = 1,2,3.5 # or implicitly using tuples (covered later)

print(x,y,z)    # print() can display multiple variables/values, separated by commas
```

```
1 2 3.5
```

To list all variables that are currently in the memory we can call the Python functions `dir()`, `locals()` and `globals()`. However, in Jupyter notebooks the IPython command `whos` gives a more readable view.

```
whos
```

```
Variable   Type     Data/Info
----------------------------
a          int      15
b          int      15
c          int      2
x          int      1
y          int      2
z          float    3.5
```

To delete a variable we can use the function `del()` or the *magic* command `%reset -f`.

```
del(x)
```

```
%reset -f
```

10.2 Data types

The following cell introduces the basic 'primitive' data types in Python, which are listed in the following table.

The Python function `type(x)` will return the data type of a variable `x`. Also the data type is displayed with the command `whos`. Notice that in the following assignments the data type is never given explicitly! Instead Python will **automatically set the data type**, that fits best to the assigned value.

```
i = 1317          # Integer (int) [e.g. -1, 0, 1, 2, 3]
f = 2.31          # Floating point number (float) [e.g. -1.0, -0.1, 0.0, 3.212, 1235.
↪213, 13.5e2]
c = 5 + 1j        # Complex number (complex) [e.g. 3 + 2j, cmath.sqrt(-1)]
b = True          # Boolean (bool) [True,False or 0, 1]
s = 'abcd'        # String (str) [e.g. 'Hello World!', 'a name', "quotation marks used␣
↪for apostrophes"]
n = None          # None (NoneType) [None]

print(type(i))    # example usage of type(). However the command 'whos' will show the␣
↪data type as well.
```

```
<class 'int'>
```

```
whos
```

```
Variable   Type        Data/Info
--------------------------------
b          bool        True
c          complex     (5+1j)
f          float       2.31
i          int         1317
n          NoneType    None
s          str         abcd
```

10.3 Naming conventions

Names must only consist of letters and numbers. Actually letters from most alphabets like Ä,ö,Ø,И,ξ,λ etc. are allowed, but it's best practice to **use only the latin alphabet and numbers**! The only valid special character is the underscore. In addition, names **must not start with a number** and **should not start with an underscore**. Usually:

- **class names** are written in 'CamelCase'
- **functions** and **variables** in 'lower_case_with_underscores'
- **constant values** are written in 'BIG_LETTERS_WITH_UNDERSCORES'.

Try to use names that are **meaningful and not longer than necessary**. Some examples are given in the following cell.

```
# GOOD NAMES
i = 1             # single letters are okay in a small context
max_value = 255   # meaningful and self explaining name
PI = 3.141        # big letters for constant values

# BAD NAMES
l = 1             # l and O are similar to 1 and 0
O = 0
λ = 7             # don't use other alphabets!
ξ = 8
print(ξ)          # even if it's valid
```

```
8
```

Note that variable names **must not equal the Python keywords**:

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

Additional information regarding naming and conventions available here.

10.4 Arithmetic operations

These are applied to numbers (integers, floats or complex). Usually we would assign the result a variable. If the result of the last expression in a cell is not assigned, it will be printed directly. We use this to make the examples cleaner.

```
1 + 3       # addition
```

```
4
```

```
3 - 2       # subtraction
```

```
1
```

```
2 * 2       # multiplication
```

```
4
```

```
4 / 2       # division (x will be a float because dividing two Integers will always␣
↪return a float)
```

```
2.0
```

```
9 // 2      # floor division (returns the quotient of the euclidian devision)
```

```
4
```

```
9 % 2       # modulo (returns the remainder of the euclidian devision)
```

```
1
```

```
2**3        # exponentiation
```

```
8
```

```
9**0.5      # root (using the exponentiation operator)
```

```
3.0
```

10.4.1 Execution order:

The execution order of operations depends on the precedence of the involved operations. Operations with same precedence will be executed from left to right (except for the 'power' operator). A detailed overview of the precedences is available here. Below two examples:

```
a = 1 + 2 * 3**2  # exponents before multiplication before addition
x = 1 / (2 * 2)   # multiplication and division have same precedence (usually left to␣
 ↪right, but parentheses used here)
print(a, x)
```

```
19 0.25
```

10.5 Boolean operations

Boolean operations are applied to booleans. The essential operations are **and (&)**, **or** (|), *xor* **(^)** and **not**.

```
b1 = True
b2 = True

# and (result is true, if both conditions are true)
b3 = b1 and b2
print('b1 and b2 = ', b3)

# or (result is true, if at least one condition is true)
b3 = b1 or b2
print(' b1 or b2 = ', b3)

# xor (result is true, if exactly one condition is true)
b3 = b1 ^ b2
print('  b1 ^ b2 = ', b3)

# not (will negate the boolean value)
b3 = not b1
print('   not b1 = ', b3)
```

```
b1 and b2 =  True
 b1 or b2 =  True
  b1 ^ b2 =  False
   not b1 =  False
```

10.6 Comparisons

Comparisons are operations that compare two values and return a boolean value.

Note that in the following examples the result of the comparison is assigned to the variable `b`:

```
b = 1 <= 5        # greater, smaller, greater or equal, smaller or equal
print(b)
```

```
True
```

```
b = 1 == 1.0      # equal (here true because same value after implicit cast)
print(b)
```

```
True
```

```
b = 'aa' < 'ab'   # comparing strings is valid (result depends on alphabetic order)
print(b)
```

```
True
```

```
b = True > 0      # booleans can be seen as 1 (True) and 0 (False)
print(b)          # -> comparison with integers is valid
```

```
True
```

10.6.1 Explicit conversions

The syntax for explicit conversions is

```
v2 = newtype(v)
```

where `v` is the variable to convert, `newtype` the new data type (e.g. `str` for string, `int` for integer or `bool` for booleans) and `v2` the variable, the result should be assigned to. Between the primitive data types, almost any conversion is valid. The following cell shows some examples:

```
f = float(123)    # integer to float
print('float(123) =', f)

f = float('5.1') # string to float (scientific notations like '2.3e3' or '-13.5e-5'␣
 ↪are also convertable)
```

(continues on next page)

(continued from previous page)

```
print("float('5.1') =", f)

i = int(141.6)   # float to integer (this will drop decimals e.g. 1.9 -> 1 and -2.9 ->
 ↪ -2)
print('int(141.6) =', i)

i = int('-2341') # string to integer
print("int('-2341') =", i)

s = str(15.3)    # float or integer to string
print("str(15.3) =", s)

b = bool(-100.3) # float or integer to boolean (every number that is not zero or␣
 ↪minus zero will be converted to 'True')
print('bool(-100.3) =', b)

i = float(True)  # boolean to float or integer ('True' will be converted to 1.0,
 ↪'False' to 0.0)
print('float(True) =', i)

c = complex(1.5) # float to complex
print('complex(1.5) =', c)

c = complex('1.5+1j') # string to complex
print("complex('1.5+1j') =", c)
```

```
float(123) = 123.0
float('5.1') = 5.1
int(141.6) = 141
int('-2341') = -2341
str(15.3) = 15.3
bool(-100.3) = True
float(True) = 1.0
complex(1.5) = (1.5+0j)
complex('1.5+1j') = (1.5+1j)
```

Invalid conversions will raise a `TypeError`:

```
#float(1.5+1j)   # this would raise a TypeError
```

Nested casts are also possible:

```
int(bool(complex(float(str(1)))))
```

```
1
```

10.6.2 Implicit conversions

At some points conversions happen implicitly e.g. when the program requires a specific data type but gets another one, or when performing operations using variables of different data types:

```
# adding float and integer will result in float (even if the result has no decimals)
f = 1 + 1.0
print("1 + 1.0 =", f)

# booleans can implicitly be converted to Floats or Integers
f = 1.5 + True
print("1.5 + True =", f)
i = 5 * False
print("5 * False =", i)

# vice versa can floats and integers also be converted to booleans if they are␣
↪connected by a boolean operation
b = -5 and True
print("-5 and True =", b)
b = not 0.0
print("not 0.0 =", b)
```

```
1 + 1.0 = 2.0
1.5 + True = 2.5
5 * False = 0
-5 and True = True
not 0.0 = True
```

Note that with both, explicid and impicid conversions we **will not not modify** a casted variable `v`, but instead create a temporary value, that represents `v` as a specific data type:

```
x = 1                # x is an integer
y = float(x)         # we assign y the float repr. of the value of x

print('x:', type(x)) # x is still an integer
print('y:', type(y)) # y is a float
```

```
x: <class 'int'>
y: <class 'float'>
```

10.6.3 Single condition

The most simple usage is the **if** statement, followed by a condition and a colon. The conditional statements follow in a block. The statements inside the block are only executed, when the condition is (evaluated to) `True`.

```
condition = True
if condition:
    print('condition is True!')
    print('still inside block!')
print('outside block!')
```

```
condition is True!
still inside block!
outside block!
```

The following example has a more complex condition which will be evaluated to a boolean value. We use the **else** keyword to introduce another block, that is only executed, when the condition is `False`.

```
x = 7

if x > 5 and not x%2 == 0:       # statement will be evaluated to a boolean value
    print('x is greater than 5 and odd!')
else:
    print('x is smaller/equal to 5 or even!')
```

```
x is greater than 5 and odd!
```

Note that all kind of blocks can be nested. For example the condition in the example above can be separated. The result of the conditional expressions can be stored to improve readability (but this also adds two more lines to read):

```
x = 7

is_greater_5 = x > 5
is_odd       = x % 2 == 1

if is_greater_5:
    print('x is greater than 5', end=' ')
    if is_odd:
        print('and odd!')
    else:
        print('and even!')
else:
    print('x is smaller/equal to 5!')
```

```
x is greater than 5 and odd!
```

10.6.4 Complex decisions

More complex structures can be built with the **elif** (short for else if) and **else** keywords. The conditions are checked, top-down untill one condition is `True`. If no condition is `True`, the else-block is executed (if it exists). The following cell demonstrates the usage.

```
cond1 = False  # some boolean variables used as conditions
cond2 = True
cond3 = False

if cond1:
    print('cond1 is True')
elif cond2:                  # will be checked if first condition is False
    print('cond2 is True, cond1 is False')
elif cond3:                  # will be checked if 1st and 2nd condition are False
    print('cond3 is True , cond1 is False, cond2 is False')
else:
    print('no condition is True')
```

```
cond2 is True, cond1 is False
```

10.6.5 Ternary if statement

This expression can be used for single line if-else statements.

```
# possible implementation to get the sign of a value
v = -3
s = 1 if v >=0 else -1
print(s)
```

```
-1
```

```
list1 = [1.0, 'name', 15, True]

print('Type:  ', type(list1))
print('Values:', list1)
```

```
Type:   <class 'list'>
Values: [1.0, 'name', 15, True]
```

10.6.6 Indexing

The elements of a list are accessed by typing the name of the variable, followed by the index in brackets. Note that the **first element has the index 0** which is usual for most programming languages but not matlab. Python supports negative indices, where -1 will give the last element of the list, -2 the second last and so on.

```
#     0    1    2    3    4    5    <- indices
L = ['a', 'b', 'c', 'd', 'e', 'f']

first_element  = L[0]   # access first element of L
second_element = L[1]   # access second element of L
last_element   = L[-1]  # access last element of L (in this case equiv. to L[5])

print('first: ', first_element)
print('second:', second_element)
print('last:  ', last_element)
```

```
first:  a
second: b
last:   f
```

Trying to access any element after the last one will **raise an IndexError**. For example our created list has currently 6 elements, so the last element has the index 5. Trying to access the element at index 6 will fail:

```
# print(L[6])       # this would rise an IndexError
```

10.6.7 Slicing

Python supports slicing, similar to matlab. Slicing means accessing a part of a list. The syntax is

`List[start:end:step]`

where `start` is the start of the slice (included), `end` the end of the slice (excluded) and `step` the step width in between. When `start` is omitted, the slice will start from the first element. Omitting `end` will make the slice end with the last element. Omitting `step` will result in a step width of 1.

```
#     0    1    2    3    4    5    <- indices
L = ['a', 'b', 'c', 'd', 'e', 'f']

print(L[1:3])      # access elements from index 1 to 3
print(L[1:5:2])    # access all elements from 1 up to 5 with step width 2
```

```
['b', 'c']
['b', 'd']
```

```
print(L[:3])       # access all elements up to index 3
print(L[3:])       # access all elements from index 3
```

```
['a', 'b', 'c']
['d', 'e', 'f']
```

```
print(L[:])        # access all elements (creates a copy of L)
print(L[::-1])     # access all elements with reversed order
```

```
['a', 'b', 'c', 'd', 'e', 'f']
['f', 'e', 'd', 'c', 'b', 'a']
```

10.6.8 Editing

Editing means changing values of a list without changing its shape.

```
list2 = [1, 1, 1, 1]
print(list2)

list2[0] = 0          # assign a new value to the first element
print(list2)

list2[2:4] = [2, 3]  # slices can be edited as well
print(list2)
```

```
[1, 1, 1, 1]
[0, 1, 1, 1]
[0, 1, 2, 3]
```

10.6.9 Manipulation

A list can be manipulated using **append**, **extend**, **del** or **remove**.

```
list2 = []               # create an empty list
print(list2)

list2.append(1)          # appends one element
print(list2)

list2.extend([2, 3, 4]) # appends a sequence
print(list2)

list2.insert(2, 0)       # inserts 0 at index 2
print(list2)

del(list2[2])            # removes the element at index 2
print(list2)

list2.remove(1)          # removes the first element that is equal to the argument..
print(list2)             # .. raises an error if argument not in list
```

```
[]
[1]
[1, 2, 3, 4]
[1, 2, 0, 3, 4]
[1, 2, 3, 4]
[2, 3, 4]
```

10.6.10 More list operations

The following cells should give an overview about the usage and capabilities of lists. In general it is better to learn what is possible first and then focus on the details (because the details can be quikly looked up).

```
L = [1, 2] * 4                        # repeats the given list 4 times
print(L)
```

```
[1, 2, 1, 2, 1, 2, 1, 2]
```

```
L = [1, 2] + [3, 4] + [5, 6]          # the + operator can be used to concatenate lists
print(L)
```

```
[1, 2, 3, 4, 5, 6]
```

```
L = [1, 2, 3]
L.reverse()                           # reverse the elements in the list
print(L)
```

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

```
L = [1, 2, 1, 3]
n = L.count(1)                          # count occurences of a value
print(n)
```

```
2
```

```
L = ['a', 'b', 'c']
b = 'a' in L                            # x in L returns true if x matches at least one␣
 ↪element in L
print(b)
```

```
True
```

```
L = [5,8,2,6,4,2.5,1]
L.sort()                                # sort list (only possible if values are␣
 ↪comparable)
print(L)
```

```
[1, 2, 2.5, 4, 5, 6, 8]
```

```
L = [5,8,2,6,4,2.5,1]
min_v  = min(L)                         # min(L) / max(L) will return minimum / maximum␣
 ↪value in L
max_v  = max(L)
length = len(L)                         # len(L) will return the number of elements in L
print('minimum =', min_v)
print('maximum =', max_v)
print(' length =', length)
```

```
minimum = 1
maximum = 8
 length = 7
```

More information about lists available here.

```
T = (False, 1.0, '2', 3)       # creation of a tuple
print('Type:  ', type(T))
print('Values:', T)
```

```
Type:   <class 'tuple'>
Values: (False, 1.0, '2', 3)
```

Once a tuple is initialized, it is neither valid to change its values nor to add/remove elements. E.g. trying to assign a new value to the first element in `T` will raise a **TypeError**:

```
print(T[0])                    # accessing elements is valid
#T[0] = 1.0                    # will raise a TypeError
```

```
False
```

Any list functionality does not modify the entries can also be applied to tuples. E.g.

- indexing / slicing
- the functions `max()` / `min()` / `len()`
- the `in`-operator.

Note: In many cases the parentheses are not required to create a tuple, but it is best practice to **use perenthesis**. Nevertheless the following cell is valid.

```
T = False, 1.0, '2', 3        # no parenthesis used to create the tuple
print(T, type(T))
print('Values:', T)
```

```
(False, 1.0, '2', 3) <class 'tuple'>
Values: (False, 1.0, '2', 3)
```

More information about tuples is available here.

```
D = {'a':1, 'b':2.0, 15:'x'}   # a dictionary with 3 key-value pairs
print('Type:  ', type(D))
print('Values:', D)
```

```
Type:   <class 'dict'>
Values: {'a': 1, 'b': 2.0, 15: 'x'}
```

The value for a specific key in a dictionary `D` is accessed using the syntax `D[k]`, where k is the key. An error will be raised if the key is not in the dictionary.

```
D = {'a':1,'b':2}
v = D['b']          # access the value that corresponds to the key 'b'
print(v)
```

```
2
```

Modifying values of existing keys and adding new key-value pairs works in the same way:

```
D = {'a':1,'b':2}
D['c'] = 3          # add a new key 'c' with the value 3
print(D)
D['c'] = 5          # change the value of key 'c' to 5
print(D)
```

```
{'a': 1, 'b': 2, 'c': 3}
{'a': 1, 'b': 2, 'c': 5}
```

The `items()` method of dictionaries can be used to iterate over each key value pair. Alternatively it can be used to convert a dictionary to a list of tuples.

```
for d in D.items():
    print(d)
```

```
('a', 1)
('b', 2)
('c', 5)
```

```
D_as_list = list(D.items())
print(D_as_list)
```

```
[('a', 1), ('b', 2), ('c', 5)]
```

More information about dictionaries available here.

A set contains **unique** elements in an **unordered** fashion (varying data types allowed). Unique means, that every element will not occur more than one time in the set. Adding a element to a set, that already contains such a element will not modify the set. Unordered means, that the elements in a set don't have indices and though indexing or slicing is not possible. Sets are created similar to dictionarys but without colons:

```
S = {1,2,3,1}     # the element 1 will only be once in S, because sets do not allow␣
↪dublicate entries
print('Type:  ',type(S))
print('Values:',S)
```

```
Type:   <class 'set'>
Values: {1, 2, 3}
```

Albeit sets don't support indexing, they allow some useful operations like computing the intersection, the union or the disjoints:

```
S1 = {'a', 'b', 'd', 1, 2, 4}
S2 = set((1, 3, 4, 'a', 'c'))   # alternative constructor (same as casting a tuple to␣
↪a set)

print('Elements in S1:               ', S1)
print('Elements in S2:               ', S2)

S_intersect = S1 & S2 # = S1.intersection(S2)
print('Elements in S1 and S2:        ', S_intersect)

S_union = S1 | S2 # = S1.union(S2)
print('Elements in S1 or S2:         ', S_union)

S_xor = S1 ^ S2 # = S1.difference(S2).union(S2.difference(S1))
print('El. in S1 or S2, not in both: ', S_xor)
```

```
Elements in S1:               {1, 2, 4, 'd', 'b', 'a'}
Elements in S2:               {1, 3, 4, 'a', 'c'}
Elements in S1 and S2:        {1, 4, 'a'}
Elements in S1 or S2:         {1, 2, 3, 4, 'a', 'd', 'b', 'c'}
El. in S1 or S2, not in both: {2, 3, 'd', 'b', 'c'}
```

More information about sets is available here.

10.7 While loop

The most simple loop is the while loop. The syntax is:

```
while cond:
    inside loop
    still inside loop
outside the loop
```

Where `cond` is a condition. The loop will repeat until the condition is evaluated to `False`. The following example runs a while-loop until there are no more elements in the list `L`. In each iteration the first element of `L` is removed.

```
L = ['a', 'b', 'c']  # create a list with 3 elements

while len(L) > 0:    # iterate as long as L is not empty (number of elements greater␣
 ↪than zero)
    print(L)         # print whole list
    del(L[0])        # delete the first element in L

print('done')
```

```
['a', 'b', 'c']
['b', 'c']
['c']
done
```

10.8 For loop

Python offers a very simple for-in syntax to iterate over each element in a collections like a list, a tuple or a set:

```
for a in B:
    inside loop
    still inside loop
outside the loop
```

Here `B` is a collection of elements (like a list, set, tuple etc.) and `a` the variable that will take the values of the elements in `B`. The indented lines after the colon are executed in every loop. In the next cell we will use a for loop to iterate over the elements in a list and print each element:

```
L = ['e1', 2, 3.5]  # creating a list
for elem in L:
    print(elem)     # elem will take the value of each element in L
```

```
e1
2
3.5
```

For loops can also be used to execute a block `n` times, having a variable that holds the number of the iteration. To create such a loop, we can use Pythons `range()` method, which will create something similar to a list:

```
n = 3
for i in range(n):     # range(3) creates the sequence [0, 1, 2]
    print('inside the loop..')
    print('i =', i)

print('done')
```

```
inside the loop..
i = 0
inside the loop..
i = 1
inside the loop..
i = 2
done
```

If `range(x)` is called with only one argument, the loop will start at 0 and incrementing up to x (not included) with a stepwith of 1. More general, one can call `range()` with up to three arguments, which denote the start, stop and step with of the sequence. Two arguments denote start and stop (step width will be 1).

```
start = 1; stop = 6; step = 2

for j in range(start, stop, step):
    print(j)
```

```
1
3
5
```

10.8.1 List comprehension

The following example demonstrates a powerful feature in Python which uses the for-in syntax called list comprehension. More information available here.

```
lst_sq = [x**2 for x in range(1, 11)] # 'for' inside brackets used to create a list
print(lst_sq)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

10.8.2 Enumeration

The **enumerate()** function is very helpful when iterating over sequences and keeping track of the index

```
L = ['a','b','c']
for index, elem in enumerate(L):
    print('Element {}: {}'.format(index, elem))
```

```
Element 0: a
Element 1: b
Element 2: c
```

10.8.3 Looping over two lists

When it is necessary to iterate over two lists simultaneously one can use the built-in **zip()** function. It takes multiple lists as arguments and creates a list-like object of touples, holding the corresponding elements of all lists. This is demonstrated in the next cell:

```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
L3 = ['I', 'II', 'III']

for x,y,z in zip(L1, L2, L3):
    print(x, y, z)
```

```
a 1 I
b 2 II
c 3 III
```

10.9 Break and continue

When using loops, there are two essential keywords: `break` and `continue`.

- `break` (immediately step out of the loop)
- `continue` (immediately start the next iteration)

The following cells will demonstrate the usage of these keywords.

```
L = []                  # create an empty List

for i in range(10):     # i will have the values [0,1,...,9]
    if i % 2 == 0:      # if i is even..
        continue        # ..jump to the next iteration
    L.append(i)         # append the current value of i to L

print(L)
```

```
[1, 3, 5, 7, 9]
```

```
# example for getting the index of a specific value
L = ['a','b','c','d']
t = 'd'
i = 0
while True:             # no condition (would run till infinity)
    if L[i] == t:       # check if we found the value
        break           # leave the loop
    i += 1              # increment search index pos

print(i)
```

```
3
```

Of course the example above is not a good way to get the index of an element, because it would result in an error, if `t` is not in `L`. Besides that, Python has already a solution for this task:

```
i = L.index(t)    # to be fair, this also raises an error if t is not in L
print(i)
```

```
3
```

```
#---------------DEFINITION OF say()--------------

def say(text):      # definition of a function with one parameter
    print(text)     # calling print()

#------------------USAGE OF say()----------------

say('hello')        # calling say()
say('world')        # calling say() again with a different argument
```

```
hello
world
```

In general, functions are used to take some arguments, do some computation with them and then return a value. Python functions will always **return exactly one value**. The following function takes two arguments and returns the absolute distance of them. We declare, which value should be returned with the keyword **return** followed by the variable name.

```
#---------------DEFINITION--------------

def distance(x, y):     # definition of a function with two parameters
    dist = abs(x-y)     # abs() is a python built-in function that returns the␣
 ↪absolute value
    return dist         # return the value of 'dist'

#------------------USAGE----------------

a = -1.5
b = 12.7
c = distance(a,b)    # calling distance() with arguments a and b. The result is␣
 ↪assigned to c
print(c)
```

```
14.2
```

You may wonder what the function say() returned, because we did not use **return** at all. By default any function will return the special value **None**. The following implementations are equivalent to the previous say() function. All of them return None.

```
def say1(text):   # first implementation, no return statement
    print(text)

def say2(text):   # 2nd implementation, return without value
    print(text)
    return

def say3(text):   # 3rd implementation, explicitly returning None
    print(text)
    return None
```

(continues on next page)

(continued from previous page)

```
r1, r2, r3 = say1('1'), say('2'), say('3') # storing the return values of all␣
↪functions

print(r1,r2,r3)
```

```
1
2
3
None None None
```

10.9.1 Multiple return types

Functions can return multiple elements by implicitly using tuples (here one usually doesn't use the parentheses to declare tuples).

```
def divide_euclidian(dividend,divisor):
    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder          # same as: return (quotient, remainder)

q, r = divide_euclidian(15, 7)          # same as: (q, r) = divide_euclidian(15,7)
print('15 / 7 = {} R: {}'.format(q, r))
```

```
15 / 7 = 2 R: 1
```

Info: Python already offers this function:

```
q, r = divmod(15,7)
print('15 / 7 = {} R: {}'.format(q, r))
```

```
15 / 7 = 2 R: 1
```

10.9.2 Keyword parameters

Functions can have optional keyword parameters. These keyword parameters are always the last (rightmost) parameters of all. They are declared by a following equal sign and the default value. The following function has one obligatory and one optional parameters. In this example the optional parameter is used as a so called 'flag' (like an on-off switch). In general keyword arguments can have any type.

```
def square(a, verbose=False):      # optional keyword parameter (here a bool)
    if verbose:
        print('square({}) was called!'.format(a))
    return a**2

sq1 = square(1)                    # nothing will be printed
sq2 = square(2, verbose=True)      # will print the message
```

```
square(2) was called!
```

Python provides many built-in functions. Some essential functions are used in the next cell. A complete list is available at programiz.com. Note that different Python versions may provide more/less built-in functions.

```
print('max(1, 9, -5) =', max(1, 9, -5))     # max(a, b, ..) returns the biggest value
print('min(-3, 3, 0) =', min(-3, 3, 0))     # min(a, b, ..) returns the smallest value
print('abs(-2)       =', abs(-2))           # abs(x)        returns the absolute value
```

```
max(1, 9, -5) = 9
min(-3, 3, 0) = -3
abs(-2)       = 2
```

Note that there are built-in functions like `del()` which are Python keywords and others like `max()`, which are no keywords. Unlike keyword built-ins, the **non-keyword built-ins can be overwritten**! This is demonstrated in the following cell:

```
# Example, that built-ins can be overwritten!

print('max:         ', max)
print('type of max:', type(max))
print('max(1,2):    ', max(1,2))
#del(max)                              # ERROR (name 'max' is not defined)

print('\noverwriting max!\n')
max = 12345

print('max:         ', max)
print('type of max:', type(max))
#print('max(1,2):    ', max(1,2))      # ERROR ('int' object is not callable)
del(max)
```

```
max:         <built-in function max>
type of max: <class 'builtin_function_or_method'>
max(1,2):    2

overwriting max!

max:         12345
type of max: <class 'int'>
```

10.10 Importing modules

We can additionally import modules like the `math` module to have access to the functionalities of that module. We can use the `dir()` function to list the available functionalities.

```
import math      # import the math module to use its features
#dir(math)      # list functionalities of a module
```

At this point we will introduce another helpful IPython feature. We can append a question mark to any module, variable, function or class, to show additional information and the documentation about it (alternatively click on a function press [SHIFT]+[TAB]):

```
math.log10?
```

```
math.pi?
```

To access / use a functionality we use the dot-operator (a dot between the module name and the function / variable).

```
x = math.log10(10000) # logarithm with base 10
print(x)

p = math.pi           # the constant pi
print(p)

y = math.cos(2*math.pi)
print(y)
```

```
4.0
3.141592653589793
1.0
```

10.10.1 Class definition

The following cell defines the class `dog`. The class contains multiple methods (methods are functions of a class).

```
class dog:                          # the definition of a class starts with the class␣
↪keyword and the name of the class
    def __init__(self, name):      # the init method defines the creation of an␣
↪instance of this class
        self.name = name
        self.age = 0               # our dogs will have two attributes: name and age.
↪'self' refers to the new dog-object

    def celebrate_birthday(self): # this method will increase the age of the␣
↪corresponding object by one
        self.age += 1

    def rename(self, new_name):    # this method takes a new name as argument and␣
↪assignes it to the dogs name
        self.name = new_name

    def __str__(self):             # definition how to cast a dog to a printable␣
↪string
        return '{}, {} years old'.format(self.name, self.age)
```

10.10.2 Class usage

Now we will create two instances of that class and use some of their methods.

```
dog1 = dog('Bonny')                # initialization of a dog. Note that only one␣
↪argument is given (the name)
dog2 = dog('Clyde')                # 'self' is always passed implicitly

dog1.rename('Bonnie')              # call of dog1s rename method with dot operator

for i in range(5):
    dog1.celebrate_birthday()
    dog2.celebrate_birthday()      # call celebrate_birthday() 5 times for both dogs

print(dog1)
print(dog2)                        # print will implicitly use our implemented string␣
↪conversion

print(dog1.age)                    # access attribute of dog1 with dot operator
```

```
Bonnie, 5 years old
Clyde, 5 years old
5
```

Additional information about classes is available e.g. at the official python documentation.

The division example could be implemented like this:

```
def divide(dividend, divisor):
    assert divisor != 0, "Division by zero not valid!"
    return dividend/divisor
```

Let us now try to call that function with an invalid divisor:

```
result = divide(1, 0)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [87], line 1
----> 1 result = divide(1, 0)

Cell In [86], line 2, in divide(dividend, divisor)
      1 def divide(dividend, divisor):
----> 2     assert divisor != 0, "Division by zero not valid!"
      3     return dividend/divisor

AssertionError: Division by zero not valid!
```

This will raise an AssertionError with our message, so we directly know what we did wrong. Maybe this is not the best example, because without the assertion the call would raise a ZeroDivisionError, which is also pretty obvious, but the general idea should be clear.

10.10.3 SyntaxError

```
y =                         # incomplete statement
```

```
  File "<ipython-input-85-fbe96dcb0c90>", line 1
    y =                         # incomplete statement
                                                      ^
SyntaxError: invalid syntax
```

```
x = (1+2) * ((3+5)/3))    # additional parentheses
```

```
  File "<ipython-input-86-7d4e5ab904ff>", line 1
    x = (1+2) * ((3+5)/3))    # additional parentheses
                           ^
SyntaxError: invalid syntax
```

```
x = (1+2) * ((3+5)/3      # missing parentheses
```

```
  File "<ipython-input-87-f684cc964df7>", line 1
    x = (1+2) * ((3+5)/3      # missing parentheses
                                                   ^
SyntaxError: unexpected EOF while parsing
```

10.10.4 IndentationError

```
if True:
x = 1                       # missing indentation
```

```
  File "<ipython-input-88-34b81d45b027>", line 2
    x = 1                       # missing indentation
    ^
IndentationError: expected an indented block
```

```
a = 1
 b = 2                      # invalid indentation
```

```
  File "<ipython-input-89-c9c618659aa0>", line 2
    b = 2                     # invalid indentation
    ^
IndentationError: unexpected indent
```

10.10.5 TypeError

```
abs()                    # calling abs() with no arguments (one argument required)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-90-db1c3457168d> in <module>()
----> 1 abs()                    # calling abs() with no arguments (one argument␣
 ↪required)

TypeError: abs() takes exactly one argument (0 given)
```

```
a,b = abs(1)             # trying to assign the result of abs() to a tuple (only one␣
 ↪return value)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-91-bfe63526b606> in <module>()
----> 1 a,b = abs(1)             # trying to asign the result of abs() to a tuple␣
 ↪(only one return value)

TypeError: 'int' object is not iterable
```

```
a = 1
a[0]                     # trying to access first element of integer (non sequence␣
 ↪object)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-92-dd699019f98a> in <module>()
      1 a = 1
----> 2 a[0]                     # trying to access first element of integer (non␣
 ↪sequence object)

TypeError: 'int' object is not subscriptable
```

```
a = 1
a()                      # parentheses indicate that a() is a method, but it is not
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-93-e083f2705fea> in <module>()
      1 a = 1
----> 2 a()                      # parentheses indicate that a() is a method, but␣
 ↪it is not

TypeError: 'int' object is not callable
```

10.10.6 ValueError

```
int('1.5')              # string cannot be converted to int
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-94-eee59e071eac> in <module>()
----> 1 int('1.5')               # string cannot be converted to int

ValueError: invalid literal for int() with base 10: '1.5'
```

```
try:
    int('1.0')              # the try block contains statements that may raise an␣
 ↪error
    a = 1/0
except ValueError:
    print('conv. failed')   # if the declared error occurs, the statements in the␣
 ↪except block are executed
except ZeroDivisionError:
    print('division by 0')  # multiple different errors can be excepted
finally:
    print("don't care!")    # the finally block (optional) contains statements, that␣
 ↪are ALWAYS executed after the try-except
```

```
conv. failed
don't care!
```

- that are only used in a small context (e.g. only in one statement)
- whose return value can be computed in one statement.

The following cell shows the usage of lambda expression to define a function:

```
def add(x, y):          # common way to define a function
    return x + y

addL = lambda x,y : x+y  # using lambda to define a function

print('Type of add: ', type(add))
print('Type of addL:', type(addL))

print('\nResult of add(1,2): ', add(1,2))
print('Result of addL(1,2):', addL(1,2))
```

```
Type of add:  <class 'function'>
Type of addL: <class 'function'>

Result of add(1,2):  3
Result of addL(1,2): 3
```

Usually lambda functions are not stored in variables, but instead directly passed as arguments. E.g. the built-in function `max()` has the keyword parameter `key` of type function. We can e.g. use this keyword, to make the function search for the biggest absolute value:

```
values = [-3, 8, -15, 2]

abs_max = max(values, key = lambda x: abs(x))
print('{} has the highest absolute value in {}.'.format(abs_max, values))
```

```
-15 has the highest absolute value in [-3, 8, -15, 2].
```

- In an interactive shell, the last result will be stored to _
- **Tripple apostrophs** introduce multi line comments, **tripple quotes** introduce multi-line strings:

```
'''
a
multi-line
comment
'''
s = """a
multi-line
string"""

print(s)
```

```
a
multi-line
string
```

- Strings are similar to tuples of single characters so **indexing and slicing can be applied to strings**

```
s = ('xyz'*5)
print(s)
print(s[:5])
print(s[-1])
```

```
xyzxyzxyzxyzxyz
xyzxy
z
```

- Conversion between different integer systems

```
x = 0xFF        # hexadecimal
print(hex(x),'=',x)
```

```
0xff = 255
```

```
b = 0b1000       # binary
print(bin(b),'=',b)
```

```
0b1000 = 8
```

```
o = 0o11         # octal
print(oct(o),'=',o)
```

```
0o11 = 9
```

- Since Python 3 integers can have **arbitrary size**

```
f = 2.0**1023   # limit of float value
print(f)

i = 3**5000     # int has no limit
# print(i)
# print(bin(i)) # simple way to produce many zeros and ones
```

```
8.98846567431158e+307
```

- Creation of a char from its **unicode number** (see unicode table) with `\u` in strings

```
lmbda = '\u03BB'
print(lmbda)
```

```
λ
```

- Creation of a char from its **ascii number** (see ascii table) with `chr()`

```
heart = chr(60) + chr(51)
print(heart)
```

```
<3
```

- Tricky **list augmentation**. Although the following four implementations have the same result, the first one is about 1000 times slower than the others (because new memory space is allocated in each iteration)!

```
L = [1,2]

L = L + [42]    # impl 1   VERY SLOW !!
L += [42]       # impl 2
L.append(42)    # impl 3
L.extend([42]) # impl 4

print(L)
```

```
[1, 2, 42, 42, 42, 42]
```

- Python supports an **else block after loops** (and try-catch blocks). That is only reached, when the while condition is not fulfilled (and not reached when the loop is terminated with the break statement)

```
i = ''
while i != 'exit':
    i = input('enter exit or break: ')
    if i == 'break':
        break
else:
    print('else block')
print('done')
```

```
enter exit or break: exit
else block
done
```

Following code-cell removes `In[]` / `Out[]` prompts left to code cells.

```
%%HTML
<style>div.prompt {display:none}</style>
```

```
<IPython.core.display.HTML object>
```

Part IV

Tutorial Principles of Programming

CHAPTER

ELEVEN

TUTORIAL NO. 1 - CHECK YOUR KNOWLEDGE

Answer the following questions with yes or no:

11.1 Basic Computer Knowledge

11.1.1 Hardware

- Do you know how disk (block devices) can be cut into partitions
- Do you know what the role of the Basic Input Output System (BIOS) is in starting up your computer
- What is the MBR
- Did you replace a component of your computer yourself (e.g., a drive, a harddisk, the graphics card)
- Did you build a computer from scratch (starting with the mainboard, adding memory and CPU, etc.)

11.1.2 The Windows Operating System

- Did you already install a Windows operating system to a computer

11.1.3 The Linux Operating System

- Have you used a Linux computer
- Have you installed a Linux computer (or virtual machine running Linux)
- Did you rent a Linux-based virtual server
- Did you write a program dedicated for running on a Linux computer

11.1.4 The Internet

- Did you use a web browser
- Did you use a command line tool for accessing the web (wget, curl,…)
- Do you know the inner workings of at least one text-based Internet protocol like SMTP, POP3, or FTP
- Do you know what RESTful services should be
- Do you know Service-Oriented Architectures (SOA) mainly built on XML-specified web services

- Did you create a web page
- Did you use web service

11.2 Spatial Data Exposure

- Do you have experience with geospatial simple feature data?
- Do you have experience with point clouds
- Do you have experience with OpenStreetMap data
- Do you have experience with RGB images (camera)
- Do you have experience with orthophotos (RGB)
- Do you have experience with optical satellite imagery (multispectral)
- Do you have experience with SAR data
- Do you have experience with mobile devices deploying some GNSS
- Do you have experience with accellerometers and gyroscopes
- Do you know projections and their impact on spatial data representation
- Are you able to reproject a GeoTIFF file into another projection

11.3 Spatial Tools

- Did you create an online map in HTML/Javascript using the Leaflet Framework
- Do you know the mapnik rendering toolkit
- Did you perform spatial statistics
- Did you perform spatial interpolation
- Did you work with QGIS
- Did you use Python within QGIS
- Did you work with Blender
- Did you use Python within Blender
- Did you work with ArcGIS

11.4 Algorithms Knowledge

- Do you know the Dijkstra Algorithm
- Do you know the Algorithm of Floyd
- Do you know how to calculate the shortest path on a sphere
- Can you imagine and algorithm to find the nearest encounter for ships on a sphere?
- Can you formulate the Traveling Salesman Problem
- Do you know what the Fréchet Distance of Spatial Trajectories is

- Do you know a means to compare sets of points with each other. If so, which one?

11.5 Special Aspects you want to see covered

(just write up to a single paragraph here)

Part V

Libraries and Stories

CHAPTER

TWELVE

READING AND WRITING PLY FILES (POINT CLOUD EXAMPLE)

It has been complicated and annoying to support a sensible subset of 3D file formats. But modern C++ comes to the rescue, see how easy the

happily reading PLY files (happly)

header-only library makes reading and working (in this case using Eigen3) with point clouds.

Source

```
#include "happly.h"
#include<algorithm>

#include<Eigen/Core>
#include<Eigen/Dense>
using namespace Eigen;


auto get_scaled_pointcloud(std::string filename)
{
    happly::PLYData plyIn(filename);
    std::vector<double> x = plyIn.getElement("vertex").getProperty<double>("x");
    std::vector<double> y = plyIn.getElement("vertex").getProperty<double>("y");
    std::vector<double> z = plyIn.getElement("vertex").getProperty<double>("z");
    double scaler=1.0;
    Eigen::MatrixXf m(x.size(),3);

    for (size_t i=0; i < x.size(); i++)
    {
        m(i,0) = x[i] / scaler;
        m(i,1) = y[i] / scaler;
        m(i,2) = z[i] / scaler;
    }


    return m;
}


int main(int argc, char **argv)
{
    auto pointcloud = get_scaled_pointcloud(argv[1]);
    Matrix3f rot;
    rot = AngleAxisf(0.25*M_PI, Vector3f::UnitX())
```

(continues on next page)

(continued from previous page)

```
      * AngleAxisf(0.5*M_PI,  Vector3f::UnitY())
      * AngleAxisf(0.33*M_PI, Vector3f::UnitZ());
    std::cout << rot << std::endl;

    auto rotated = (rot * pointcloud.transpose()).transpose();

    for (size_t i=0; i< 10; i++){
       std::cout << "X:"<<(rot * pointcloud.row(i).transpose()).transpose()  <<␣
↪std::endl;
       std::cout << "Y:" << rotated.row(i) << std::endl;
    }

    std::vector<double> x,y,z;
    x.resize(rotated.rows());
    y.resize(rotated.rows());
    z.resize(rotated.rows());
    std::cout << "Warper" << std::endl;
    for (size_t i=0; i < rotated.rows(); i++)
    {
        x[i] = rotated(i,0);
        y[i] = rotated(i,1);
        z[i] = rotated(i,2);
    }
    happly::PLYData plyOut;

   plyOut.addElement("vertex", x.size());

   plyOut.getElement("vertex").addProperty<double>("x", x);
   plyOut.getElement("vertex").addProperty<double>("y", y);
   plyOut.getElement("vertex").addProperty<double>("z", z);

   plyOut.write("out.ply", happly::DataFormat::ASCII); // or Binary

   return 0;
}
```

Makefile

```
all:
        g++ -Wall -march=native -Ofast -o libs_pcl_happly libs_pcl_happly.cpp  `pkg-
↪config --cflags --libs eigen3`
run:
        ./libs_pcl_happly ../data/sofa_0007.off.ply
```

Build Output

Run Output

CHAPTER

THIRTEEN

HAPPLY AND BOOST GEOMETRY - LOAD POINT CLOUDS IN C++11

The happly library is a nice header-only library for reading and writing PLY files. The Boost Geometry library is an efficient OGC Simple Features comaptible spatial computing framework.

They can be bound together nicely as illustrated in the following snippet:

Source

```
/** \file Implementations related to Distance matrix

*/


#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/index/rtree.hpp>
#include <boost/geometry/algorithms/distance.hpp>


#include<numeric>
#include <random>

#include<fstream>

#include "happly.h"
#include "tictoc.hpp"

namespace bg = boost::geometry;
namespace bgi = boost::geometry::index;


using point=bg::model::point<double, 3, bg::cs::cartesian>;
using pointcloud = bg::model::multi_point<point>;


  /**
  *   @brief  Loads a PLY file and scales (aspect-correct into the unit cube).
  *
  *   @param  filename is the name of the file, at the moment PLY is supported
  *   @return the pointcloud
  */
```

(continues on next page)

(continued from previous page)

```
pointcloud get_scaled_pointcloud(std::string filename)
{
    happly::PLYData plyIn(filename);
    std::vector<double> x = plyIn.getElement("vertex").getProperty<double>("x");
    std::vector<double> y = plyIn.getElement("vertex").getProperty<double>("y");
    std::vector<double> z = plyIn.getElement("vertex").getProperty<double>("z");
    // translate to origin
    const auto minx = *std::min_element(x.begin(), x.end());
    for (auto &ix : x)
       ix  -= minx;
    const auto miny = *std::min_element(y.begin(), y.end());
    for (auto &iy : y)
       iy  -= miny;
    const auto minz = *std::min_element(z.begin(), z.end());
    for (auto &iz : z)
       iz  -= minz;
    // scale to unit cube
    auto scaler = std::max ({
                      *std::max_element(x.begin(),x.end()),
                      *std::max_element(y.begin(),y.end()),
                      *std::max_element(z.begin(),z.end()),
                      });
    pointcloud m;
    m.resize(x.size());

    for (size_t i=0; i < x.size(); i++)
    {
        m[i] = point{x[i] / scaler,y[i] / scaler,z[i] / scaler};
    }

    return m;
}

  /**
  *   @brief  Instantiates an affine matrix for rotation around the X axis.
  *
  *   @param  angle in radians
  *   @return matrix_transformer (the matrix is accessible from .matrix())
  */
bg::strategy::transform::matrix_transformer<double,3,3> xrot(double angle)
{
    return bg::strategy::transform::matrix_transformer<double, 3, 3> (
            1,0,0,0,
            0,cos(angle), sin(angle), 0,
            0,-sin(angle),  cos(angle), 0,
            0,0,0,1);
}


  /**
  *   @brief  Instantiates an affine matrix for rotation around the Y axis.
  *
  *   @param  angle in radians
  *   @return matrix_transformer (the matrix is accessible from .matrix())
  */
```

(continues on next page)

(continued from previous page)

```
bg::strategy::transform::matrix_transformer<double,3,3> yrot(double angle)
{
    return bg::strategy::transform::matrix_transformer<double, 3, 3> (
            cos(angle),  0.0, -sin(angle), 0.0,
            0,1,0,0,
            sin(angle),  0,cos(angle),   0.0,
            0,0,0,1);
}

  /**
  *   @brief  Instantiates an affine matrix for rotation around the Z axis.
  *
  *   @param  angle in radians
  *   @return matrix_transformer (the matrix is accessible from .matrix())
  */

bg::strategy::transform::matrix_transformer<double,3,3> zrot(double angle)
{
    return bg::strategy::transform::matrix_transformer<double, 3, 3> (
            cos(angle), -sin(angle),  0.0, 0.0,
            sin(angle),  cos(angle),  0.0, 0.0,
            0.0,          0.0,        1.0,  0.0,
            0,0,0,1);
}

// let us apply a rotation
namespace trans = bg::strategy::transform;



  /**
  *   @brief  Returns a rotated copy of the given point cloud given three angles for␣
 ↪X, Y, Z
  *
  *   @param in is the point cloud to be copied from
  *   @param  xAngle is the angle around the X-axis in radians
  *   @param  yAngle is the angle around the Y-axis in radians
  *   @param  xAngle is the angle around the Z-axis in radians
  *   @return the rotated point cloud
  */

pointcloud rotated(const pointcloud &in, double xAngle,
double yAngle,double zAngle)
{
   bg::strategy::transform::matrix_transformer<double,3,3>
    tr (
        xrot(xAngle).matrix() *
        yrot(yAngle).matrix() *
        zrot(zAngle).matrix()
        );
  pointcloud dst;
  boost::geometry::transform(in, dst, tr);
  return dst;
}
```

(continues on next page)

(continued from previous page)

```
  /**
  *   @brief  Stores a point cloud into a binary PLY file.
  *
  *
  *   More concretely, it takes the given point cloud, creates vectors (memcopy,␣
 ↪inefficient,
  *   but at the moment okay) and for each of those vectors creates an element "vertex
 ↪"
  *   with attributes X, Y, and Z holding double values as typical.
  *
  *   @param  filename is the name of the file
  *   @param  pcl is the point cloud
  *   @return void
  */

void writePly(std::string filename, const pointcloud &pcl)
{
    std::vector<double> x,y,z;
    x.reserve(pcl.size());
    y.reserve(pcl.size());
    z.reserve(pcl.size());
    for (const auto &p:pcl)
    {
        x.push_back(bg::get<0> (p));
        y.push_back(bg::get<1> (p));
        z.push_back(bg::get<2> (p));
    }

    happly::PLYData plyOut;
    plyOut.addElement("vertex", x.size());

    plyOut.getElement("vertex").addProperty<double>("x", x);
    plyOut.getElement("vertex").addProperty<double>("y", y);
    plyOut.getElement("vertex").addProperty<double>("z", z);

    plyOut.write(filename, happly::DataFormat::Binary); // or ASCII

}


template<typename benchmark_strategy=notictoc>
std::vector<double> distanceMatrix(const pointcloud &A,const pointcloud &B)
{
    std::vector<double> dm (A.size() * B.size());
    { benchmark_strategy bench("DistanceMatrix for "+std::to_string(A.size()) + " and
 ↪" + std::to_string(B.size()));
    for (size_t i=0; i<A.size(); i++)
    {
      dm[i*B.size()+i] = 0;
      for (size_t j=i+1; j<B.size(); j++)
        dm[j*B.size()+i]=dm[i*B.size()+j] = bg::distance(A[i],B[j]);
    }
    }
    return dm;
}
```

(continues on next page)

(continued from previous page)

```
namespace strategy{
/// The strategy for taking the mean of the distance matrix
  class mean{};
};


  /**
  *   @brief  Compute Distance with given strategy
  *
 * @param A a point cloud
 * @param B a point cloud
  *   @return the distance
  */
template<typename strategy>
double pcl_dist(const pointcloud &A, const pointcloud &B){
    throw std::runtime_error("Not specific enough. Give a distance");
    return 0;
}




/**
 * @brief Implement the distance using mean strategy
 *
 * @param A a point cloud
 * @param B a point cloud
 */
template<>
double pcl_dist<strategy::mean>(const pointcloud &A, const pointcloud &B)
{
    auto dm = distanceMatrix(A,B);
    double v = std::accumulate(dm.begin(),dm.end(),0);
    return v / dm.size();
}
```