Contents of Presentation

Chapter 17 Programming languages and tools 305

Programming as an art form 305

Donald Ervin Knuth 306

Algorithm 307

Interpreter without leeway 307

Programs that write programs 309

Augusta Ada King, Countess of Lovelace 310

Translation tools 311

A colorful bouquet of programming languages 315

Functional programming languages 316

Object-oriented programming languages 317

Logical programming languages 319

Chapter 29 Python for beginners 495

Python and other snakes 495

Snake worlds 497

Python basics 497

Our first (real) Python program 498

Different types 499

Range 502

Mathematical operators in Python - overview 503

The most important data types in Python 504

Fixed data types in Python - Overview 505

Lists 506

Help function in Python 507

Tuple 508

Sets (set) 508

Dictionaries (dict) 509

Chapter 30 The colorful wide world of Python 511

Comprehensive understanding 511

Putting characters in chains 513
Sparking functions 516
Exceptions in Python 519
Generators and factory functions 520
Oh dear - now it's all about OOP 521
Underscores in Python 522
Chapter 31 Luring Python out of its basket 527
Manage modules 527
NumPy for home use 529
Fancy indexing 529
Slicing 529
Vectorization 532
Masking without carnival 533
Create graphics with Matplotlib 533
Serializing objects with Pickle 537
Beyond the edge of the plate 538
Chess 542
...and even more fun! 543
Chapter 32 Becoming a snake charmer 545
Time measurements 545
C extension of Python 547
The fun begins 550
The byte code 551
Speed of light with vectorization 552
Chapter 41 Guided tour through the evidence chamber 677
On the trail of cyborgs 677
Knowledge without conscience 680
Planning and decision-making 680
Pattern analysis and recognition 681
Intelligent agents or search or what? 681
Artificial beings with their own consciousness 682
42 Searching and finding through play 684

Tracking with GPS 684
The mountaineering method 687
Heuristic search in the hay 690
Navigating to the stars with the A* algorithm 693
Fun with MINIMAX and Moritz 694
Pruning from alpha to beta 698
Chapter 43: Noisy systems 703
Machine learning 703
Inference without excuses 705
Landing on the knowledge base 706
inductive and deductive methods 706
Noise in the data forest 707
Outsmarting the human perception of temperature 707
Learning with a concept 707
Learning to decide with trees 712
Borderline considerations… 715
Learning without a teacher 718
Chapter 44 Expert systems for professionals 721
Prologue 721
Expert knowledge 724
Diagnoses from the electronic brain 727
Case-Based Reasoning. 728
Make predictions and get rich 734
Chapter 45 Artful neural networks 737
Copying is better than studying 737
Forward to the interconnected networks 740
Rosenblatt's Theorem 742
Rules for Learning 742
The XOR problem 745
Progress through Backpropagation 746
Gradient method 747
Squeeze me! 748

Derivation of the Error Function 750
Weight adjustment of a neuron in the output layer. 752
Weight adjustment of an internal neuron 752
Various Variants 753
The Power of Feedback Loops 754
Limitless fields of application 756
Chapter 47 Designing and creating on the web 759
Web technology for insiders 759
HTTP in short form 760
HTTP status codes 762
HTML in short form 763
HTML to XML 763
Deep Web, Darknet... 768
Chapter 48 Scripting languages 769
Scripted shell scripts 769
Not a bit cumbersome: AWK 773
Diving for pearls with Perl 775
The triumph of PHP 778
JavaScript 779

Chapter 17 Programming languages and tools

Learning to program means acquiring a manual skill.

It requires effort and perseverance, but in the end the conviction that a good program is like a work of art prevails.

Programming as an art form

I have heard people say that they could never understand programming. Just as most people are unable to create a painting that corresponds to their own ideas. In both cases, however, it is your own will that is decisive.

Learning the respective craft skills requires time and patience. Where artistic talent is at the forefront of painting, you will need to import

Programming requires a strong sense of logical thinking and a certain degree of abstraction, and the fun comes naturally.

However, I am by no means suggesting that programming is not an art, on the contrary. Probably the most important work in this field is called The Art of Programming by Donald Knuth.

Donald Ervin Knuth

Donald Ervin Knuth was born in Milwaukee, the largest city in the state of Wisconsin, in 1938. He was regarded as a child prodigy at an early age and certainly made his school very happy with his achievements.

For example, in a language competition where he had to find different words from the letter combination of 'Ziegler's Giant Bar' (the sponsor was a confectionery manufacturer called Ziegler), he found many more solution words than the judges themselves. He won a television for school and a chocolate bar for all his classmates.

At the age of thirty, he became a professor of computer science at Standford University. During this time, he also wrote the first part of his famous work, which was originally only intended to cover compiler construction, but over time developed into a comprehensive compendium on programming in general. To date, four volumes have been published, all of which are suitable as a knowledge base for computer scientists.

Incidentally, the title of his most important work is no coincidence. Knuth attached great importance to aesthetics right from the start. He developed the TeX (pronounced 'tech') typesetting system specifically for his work on the mein In work, which can be used to present any text in an attractive format. Leslie Lamport further developed the system with macros to create LaTeX, which is still widely used and popular today, especially in the academic world.

Nevertheless, there is a difference between painting and programming, and it is somewhat more deeply hidden. In painting, the idea of the final work can fail due to one's own possibilities, but not in programming. On the contrary, the idea of the final product is a prerequisite for a successful programme. As long as this idea is a fantasy product and there are only vague outlines of the goal, a programme is not in sight.

An example of an unclear description would be: 'I am now writing a programme so that I can talk to my computer like a human being!'

What I mean by 'idea of the final work' is the algorithm! This already contains pretty much everything you need for a successful programme, even though not a single line of code has been written. Worse still, you haven't even decided on a programming language yet.

Algorithm

The word algorithm is used as a synonym for a programmable computational rule. It goes back to the name of the Persian mathematician "al-Khoarizmi", which is difficult to pronounce. As early as 1825, he published the extremely important work on arithmetic with Indian numerals, in which he introduced the operation with the symbols we still know today, the decimal digits, which are incorrectly called Arabic numerals. Today we use the word algorithm much more generally as a precise representation of systematic calculation. Unfortunately, hardly anyone thinks of "al-Khoarizmi".

Instead of comparing a program to a work of art, compare it to a building, such as a skyscraper. It is important to plan utility shafts, elevators, and stairwells early on. In a way, the algorithm is the blueprint - it gives you a very precise idea of what the final building will look like. This requires craftsmanship. But when you are ready to immerse yourself in this world, programming gives you a sense of happiness that makes you forget time and space.

Interpreter without leeway

At first glance, this sounds like a lot of work - as we all know, it takes a lot of time and effort to learn a craft. Wouldn't it be easier to have a program running all the time on your computer that executes the programmer's commands immediately? The concept actually exists. It is called an interpreter.

The interpreter does not translate the program into machine code first, but executes the instructions in the source code directly. It is easier to understand the difference between a classic compiler and an interpreter if you look at runtime and compile time.

The runtime of a program is the time during which its code is executed by the computer. At compile time and translation time, the code of a program is converted into machine code. At compile time, the computer does not execute the code of the program itself, but some code of the compile

If you work with a compiler, the source code of your program is first compiled once. You then have access to the machine code of your program, which you can execute as often as you like. You only need the compiler at compile time, after that you no longer need it. Theoretically, you could delete it immediately afterwards. Your program would not care. However, I would not recommend this: after all, you want to compile other programs later!

If, on the other hand, you start your program from an interpreter, it will immediately take over the execution of your source code. This means that no machine code is generated at all. However, you will of course need the interpreter to run your program, otherwise the code could not be executed.

Think of the interpreter as your own computer, a virtual machine that immediately understands the source code of your language. In most cases, a certain amount of revision of your program will be necessary before it can actually be executed, a kind of pre-processing. This could include a syntactic analyst, the creation of a table of variables and their types and a list of jump labels.

In principle, it does not matter which programming language you use Theoretically, any programming language can be compiled or interpreted.

The advantages of an interpreter are also the disadvantages of a compiler and vice versa. I would just like to give you a few arguments to make it easier for you to choose between compilers.

- The interpreter consumes resources at runtime, so a compiler is preferable where the speed of program execution or optimum use of memory space is important.

- You can test interpreted programs before they have been fully developed. They run immediately and do not require recompilation for every tiny change.

- It seems pointless to repeat a necessary syntactic analysis, as interpreters and compilers - of course - have to do, every time a program is executed. In this sense, compilers are preferable to interpreters.

- The main reason why interpreters still play an important role today is another: errors in interpreted programs are much easier to detect. The interpreter generally gives clear and unambiguous indications of when and where something has gone wrong. This is more difficult with compiled programs. The error may only occur at runtime, by which time the compiler has finished work. Finding the exact location is also much more complex because the compiler may have optimized the structure of the program for translation. This means that the result no longer corresponds exactly to your original source code. To compensate for this, the compiler deliberately inserts additional machine code for error correction, known as debugging. This allows to localize problems more precisely,

, find original variable names again and provide clearer error information. As soon as the program is finally ready for delivery, this debugging information is removed again, stripped.

In fact, the transition between interpreters and compilers is fluid. Your program could be translated by a compiler, but not into machine code, but into the code of a virtual machine. This is called bytecode, by the way. This would then be interpreted and ultimately executed.

Conversely, there are just-in-time compilers, JIT compilers. These first translate - like an interpreter - at runtime, but then immediately into machine code.

Historically, there have been and still are a vast number of programming languages that were either compiled or interpreted. In addition, as already mentioned, both compiler and interpreter solutions have been developed for important programming languages. At the end of this chapter, I will introduce you to a number of famous programming languages in more detail.

Programs that write programs

Basically, interpreters and compilers play an intermediary role between the programmer and the computer. They are able to convert code that the machine does not immediately understand from a more human representation into one that the hardware can cope with.

So why is a program necessary at all? Can't the end user just communicate with the computer the way they want to? In principle, yes, but a computer is a universal instrument. Writing a program that allows simple, precise and comprehensible communication with the user for any purpose is a tremendously difficult task.

That's why there are so many different programs. We do not assume that the user has any specific knowledge of a computer. Depending on the intended use, the operation of a program, an application, must be immediately obvious. The adjective intuitive has become established for this purpose. You can recognize intuitive user and menu guidance by the fact that something is missing or at least superfluous: the operating instructions. The optimal program is so logically structured, so comprehensible, so clear and unambiguous that it needs no introduction, no explanation, not even a manual.

Conversely, of course, this poses a particular challenge for the programmer. They need to know how a microprocessor works in principle and what goes on inside the computer. None of this should be of interest to the user of the program, on the contrary: spare them the trouble!

The crucial question is therefore: How do you create a program that meets these requirements?

Interestingly, this central topic was already dealt with a very, very long time ago. Long before computers even existed, by the way. And it was a woman who came up with pioneering solutions. Her name was Ada Lovelace and she was the very first programmer.

Augusta Ada King, Countess of Lovelace

The mathematician Ada Lovelace was born in London in 1815 as the daughter of Lord Byron and his wife Anne Isabella. In 1835 she married Byron William King, who became the first Earl of Lovelace a few years later. Both her mother and her husband encouraged Ada's mathematical interests. This was also necessary, as women at the time were denied access to libraries, for example

Due to her zeal and enormous talent for mathematics, she soon became interested in the Analytical Engine, a forerunner of the modern computer, which had been described - in theory - by Charles Babbage.

Ada Lovelace developed the first algorithm for the Analytical Engine and is therefore rightly regarded as the very first female programmer.

It is very instructive what she thought about the basic possibilities of this machine: “The Analytical Engine has no ambition to create anything. It can do anything we tell it to do.

Two aspects of this statement are particularly important. Firstly, Ada emphasizes that the Analytical Engine has no motivation of its own to do anything. This may be obvious to today's readers, but in the 19th century some people considered such machines to be beings with independent will. Due to recent advances in the field of AI, this idea has become relevant again even today. On the other hand, the special significance of programming is emphasized. The computer can do anything that we know how to tell it to do. This is the key message for all programmers: any kind of intelligent data processing is possible as long as we define exactly how it should work.

Ada Lovelace died of cancer at the age of 36. The programming language Ada as well as numerous schools, institutions and projects, especially for girls and women, are named after her.

There are no limits to your imagination when it comes to creating programs. For example, why not simply develop a program that in turn invents other programs? You can! You can also develop algorithms for programs that are responsible for developing programs. Anything is possible, you just have to keep the constraints in mind:

- The code you develop yourself must be written in an existing programming language

- You can use as many compilers or interpreters as you like.
- Each program is a sequence of individual commands.
- In the end, your program runs on a processor that essentially only executes arithmetic-logical operations.

But don't worry! There are a number of tools that will make your life easier and help you achieve your programming goals.

Translation tools

Mastering a programming language is much quicker and easier than learning a foreign language. This is because a computer can only perform certain operations, whereas a human language has almost unlimited expressive power. What a computer language has to say is basically limited to the following categories:

- arithmetic-logical operations

- jump instructions, conditional or unconditional

- subroutine calls (including system calls)

- Assigning values to variables (memory locations)

- Retrieving values from variables (memory locations)

The last two points describe the transport of data. This also includes input and output operations. They therefore correspond to the different variants of the mov command in assembly languages.

Don't understand mov? Take a look at assembly language programming in Chapter 16 to find out!

In this chapter, I have not yet bothered you with the basics of subroutines, but this is now unavoidable. Subroutine calls are a central component of all programming tasks. They are an enormously important method for the programmer to deal with the complexity that arises.

Imagine you are writing a program to manage your stamp collection. In addition to entering new stamps and various functions for editing the country of origin, year of issue and value of the individual items, it would also be interesting to sort them.

However, there are different ways to do this, for example by...

- age,

- price and

- origin.

If you really had separate program code for each of these options, it would be much more logical to provide the Sort function with a parameter that describes either the age, the price or the origin.

A parameter of a subroutine corresponds to the variable of a mathematical function. The terms subroutine, procedure, routine and function are often used synonymously. The same applies to the terms parameter and argument for the input values. Strictly speaking, the parameter is part of the declaration of a subroutine, while the argument refers to the value with which the subroutine is called at runtime (this value can also be a reference to a memory object). You will find a more detailed analysis of this at the end of Chapter 18,

You would therefore always pass a concrete value as an argument to your sorting function at runtime, namely “age”, “price” or “origin”.

The advantage of this approach is immense:

- You only have to create the code for sorting once.

- Every change affects all sorting options at the same time.

- The final version of your program has a smaller scope

However - not a rose without thorns - there is also a downside:

- The code of the sort function with parameters is more difficult to program if you had decided in advance on a specific value in each case.

However, the bottom line is that creating subroutines is a must. Realize that this function may not even have to come from you. If you use functions from collections of subroutines, so-called libraries, your code will end up being created much faster, much shorter, easier to check, test and maintain.

Fortunately, programs have not only been written since yesterday For most standard tasks, there are now ready-made solutions in libraries, including for the sorting described in the example. The important algorithms for this are examined in Chapter 35.

Due to the limited nature of the computer's basic operations, you only need to understand a few concepts within the framework of a programming language, and there are even a number of tools available to help you learn programming even faster. In particular, editors with which you write your programs, the so-called source code, are getting better and better

You will find the following features in the editors, which go beyond ordinary word processing:

- Syntax highlighting. This means that the key words of the respective program language are highlighted in color

- Pretty-Printing. The indentation is done in a way that makes the code clearer. As we all know, the eye also helps to program. The Python programming language even requires this syntactically!
- Auto-completion. As you type, your editor guesses what you want to write and completes the entire word for you after the first few letters of a long identifier, for example.

After editing the source code, the compiler is called. It translates the source code into the object code of the machine. In some cases, the compiler also generates code that must first be translated from an assembler into the object code.

Larger programs consist of more than just a single file. The compiler generates a separate object file from each individual source file. This is quite practical so far, because changes only affect one file at a time. In the end, however, we all agree that a common object file must be created. The linker, the program connector, is responsible for this. The linker not only adjusts the addresses of the program code of the individual files, but can also resolve cross-references. Just imagine that a variable is defined in file A that is used in file B. Only the linker ensures that this external reference is replaced by the correct memory address. Of course, you can also use functions from external libraries. The linker guarantees that the associated code is actually available to your program at the end. It is important to note the difference between dynamic and static linking of such libraries.

In principle, code from libraries can be integrated into your program in two different ways, either statically or dynamically. With static binding, the complete code of the required subroutines is included as part of your own program. This makes your program considerably more extensive. In contrast, dynamic binding expects the required code to be made available to your program at runtime in the form of separate files.

The advantages of dynamic linking are obvious: Your program remains much leaner, and your computer only has to provide the relevant code once for all programs that use the same dynamic libraries.

The disadvantage of this procedure usually becomes apparent when you run the corresponding programs on another computer. If the corresponding dynamic libraries are not available there, or only in the wrong version, this leads to unsightly error messages.

A dynamically linked library is called a dynamic link library, dll. A static library is abbreviated to lib for static library. The executable file is an executable, or exe for short. I have illustrated the difference between static and dynamic linking a little more clearly in Figure 17.1

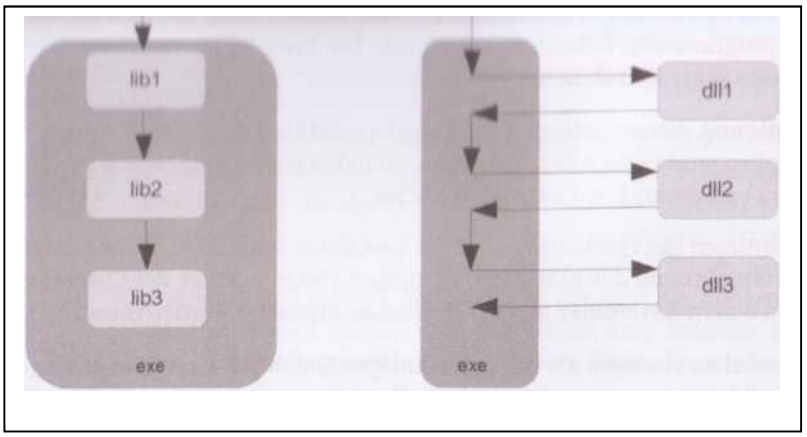


Figure 17.1: Difference between static (left) and dynamic binding (right)

A brief summary of the steps required to turn a source code into a single file can be found in Figure 17.2.

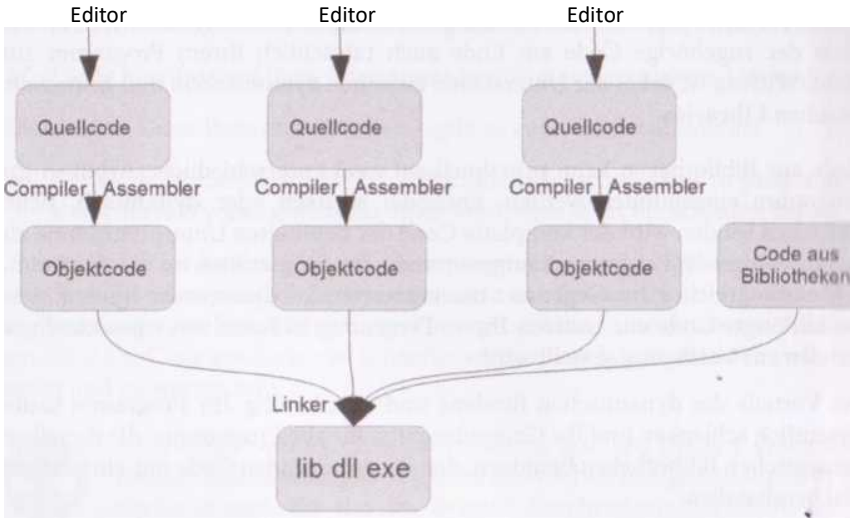


Figure 17.2: Steps for program creation

Use the editor to create your source code files, each of which is divided into object code and passed to the linker. Together with code from external libraries, this generates coherent machine code. This is either an independent executable file or is available as a library function to other programs for dynamic or static linking.

However, the moment your program, which is located on a peripheral memory, is to be started, someone must adjust all addresses.

This is because at this moment, the programs does not yet know to which exact address in the main memory it will later be loaded. A large number of programs are constantly running on a computer and the start address can change each time it is restarted. In addition, the required memory segments must first be requested. This is the task of the loader It loads the program. Depending on the complexity of the operating system, other components may also come into play, but we can leave these out for now. The loader is part of the operating system, not your program!

However, I have withheld the best part from you so far. Nowadays, a decent programming language comes with an integrated development environment (IDE), which takes care of all the steps for you at the click of a mouse! This means that there is hardly any excuse not to start programming straight away.

A colorful bouquet of programming languages

After these rather theoretical considerations, I would like to introduce you to a few programming languages at the end of the chapter. In order not to lose the overview in this wild jungle of possible alternatives, I will present individual representatives of the most important classes, each of which supports a common programming style and thus follows a programming paradigm.

Imperative and declarative programming languages

Imperative programming languages are command-oriented.

Think of a Roman emperor like Caesar or Nero! Your commands are processed one after the other, without queries or your computer's own considerations. Imperative programming is the closest to hardware. For example, all assembly languages follow the imperative programming paradigm. Other important representatives are:

- Fortran (for formula translation) appeared as early as 1957 and dominated the world of high-level languages for a long time, especially in technical applications.
- Pascal (named after the French mathematician Blaise Pascal) was developed by the Swiss computer scientist Nikolaus Wirth specifically for the purpose of teaching. He wanted to promote structured programming in particular. He later invented Modula.
- Although Ada (named after Ada Lovelace) was not published until 1983, it is considered to be the first standardized programming language. A prominent user of the language was the US Department of Defense.

- Cobol (for Common Business Oriented Language) has been used since 1959, after numerous further developments, primarily in the economy to this day. The orientation towards human language was used early on to differentiate it from a more technical orientation, as is the case with Fortran, for example.
- C is probably the most important imperative programming language to date. It was developed by the New Yorker Dennis MacAlistair Ritchie as the successor to the B language. Ritchie used it in the 1970s for hardware-related programming. In particular, important operating systems such as UNIX (in all its derivatives, including Linux) are written in C. C is also a high-level language, but allows direct access to system resources. The C language is extremely lean and easy to learn. This probably explains its success. Part V of your book deals exclusively with C and its various offshoots.

The opposite of imperative, in terms of programming paradigms, is declarative

With imperative programs, the programmer has absolute control at all times over which commands are executed in which order. In contrast, declarative programs are descriptive. The programmer only says what is to be done and no longer how.

Take, for example, the control of a mobile robot through a room. An imperative programming language would clearly define each individual step of the machine at any given time. Of course, the program flow might also contain branches that depend on certain conditions. Nevertheless, the programmer would be responsible for every detail of the movement through the room.

With a declarative programming language, the programmer would only specify the destination. The route would have to be determined by the program flow (usually implemented using an interpreter). Here, too, the programmer could intervene, but would also have less control over the results of his input. At first, this sounds too good to be true: let's just let declarative programs determine everything! But it's not that great. It is particularly important for the programmer to know how inputs are processed so that he can achieve his goal in the end.

You can imagine the pinnacle of declarative programming languages as simply explaining to Siri, Cortana or Alexa what you want. The electronic helpers will then do the rest. Of course, you are also welcome to try out the limitations of this idea for yourself ...

Functional programming languages

Functional programming languages make greater use of the KMJ of mathematical functions than imperative programming languages. In particular, they avoid side effects that are difficult to understand, i.e. the effects of a function on memory areas that are not themselves part of the output.

The most important representatives of functional. programming languages include*.

- Lisp (for list processing) was published one year after Fortran. Fortran, namely in 1958 at the MIT (Massachusetts Institute of Technology). It is a realization of the lambda calculus, a formal language of theoretical computer science. However, Lisp is by no means limited to purely academic applications. As a comprehensive programming language, any problem can be tackled with Lisp.

- Scheme can be described as a Lisp derivative, even if it also fulfills imperative programming paradigms. Scheme was developed in 1975, also at MIT, and is considered “a simple, modern Lisp”. Since only very few language constructs are specified in Scheme, this language can be learned quickly and extended easily. Scheme has actually also been used in some industrial areas, especially where artificial intelligence concepts are required.

Object-oriented programming languages

Object-oriented programming, abbreviated OOP, understands the 'Weh. as a collection of objects that communicate with each other. This is why you can create a whole hierarchy of classes in OOP, whose properties are inherited and whose instances send messages to each other.

The basic idea of OOP emerged in the 197Os with the first graphical user interfaces. The idea is that the individual graphical objects such as buttons and menus are no longer simply processed one after the other, but that they are simply waiting for the user to click on them. Clicking in turn generates a message that is sent to the associated object. By processing a whole cascade of nested classes, the code associated with this button is ultimately executed imperatively. This code is encapsulated in the object of the button, so to speak, and does not interfere with the code of other objects.

This may sound a little complicated, but it is much simpler than constantly running through loops that query all objects one after the other for possible activities. Don't forget a button!

However, OOP is by no means limited to graphical objects. You can define any classes for all conceivable areas. Nowadays, OOP is the dominant paradigm of high-level languages. The most important representatives are:

- Smalltalk is a purely object-oriented programming language in which even elementary data types such as numbers and characters are realized as objects. The Xerox PARC research center released the first version as early as 19T2. The language still exists today and serves as a template for more or less all other OOP languages. However, its distribution is limited. The important concept of the Model View Controller (MVC) was already implemented in Smalltalk and later adopted by many programming languages.

- C++, C# and Objective-C are extensions of C that have adopted some concepts, but at the same time continue to have imperative elements. Depending on the programmer's preference, they can use the advantages of object orientation or continue to work close to the system. In contrast to Smalltalk, which poses syntactic challenges for imperative programmers, classic C programs can also be largely translated with the compilers of C derivatives - meaning that the complete C code of all libraries can in principle also be used there. A decisive market advantage!
- Java was introduced by Sun Microsystems in 1995 and was originally a child of the rapidly expanding Word Wide Web. The idea was to develop the simplest possible cross-platform code that could even run on embedded systems in televisions, refrigerators and washing machines. Even if the latter did not necessarily materialize as the developers had imagined, Java is clearly a dominant language on the Internet today. In addition, there are countless stand-alone applications that are no more closely connected to the WWW than any other language. Java has been influenced by Smalltalk and the object-oriented C derivatives. Part VI of your Di book is dedicated to Java!
- Python occupies a special place in the list of programming languages because it is an interpreted OOP language, but can also be compiled and even used as a scripting language. Being particularly easy to learn, Python is increasingly proving to be an all-purpose weapon in numerous areas of computer science, including the increasingly important field of machine learning. Today, it is one of the most popular programming languages of all, if the Dutchman Guido van Rossum Python already sun it in the 1990s, its triumphal march since version 3 (end of 2008) seems unstoppable. You can find more details on this and everything important about Python in Part VII.

Swift is the youngest of the languages presented here. It was presented at an Apple developer conference in 2014. Originally designed for the IOS world PF, Swift is now preparing to make its triumphal march through the rest of the programming world as open source. It has produced a few new features such as **Optionals** or **Tuples**, is very intuitive and requires surprisingly little code. You can find more information on Swift in chapter 20.

- Almost all classic programming languages that are still used today now use the important OOP concepts. These include not only Fortran or Cobol, but also Lisp and even Ada

Logical programming languages

Logical programming languages are the other extreme of imperative languages. A logical programming language provides the computer with a set of initial statements and a set of allowed rules. The logical interpreter then has to find out which statements are valid under exactly these conditions and which are not. Logical programs are based on the idea that the computer simply calculates the 'truth content' of statements. Numerous concepts have been developed for this purpose.

Logical programming languages have not been able to gain widespread acceptance, although they are still used today not only in purely academic fields, but also in database systems and in some areas of artificial intelligence.

Chapter 29 Python for beginners

The most popular programming language at the moment is called “Python”. In this chapter, you will find out why this is the case and I will also explain why this computer language has - apparently - taken on the name of a snake. This is not true at all. Nevertheless, you will certainly be fascinated by this fascinating language.

This chapter is not intended as a general introduction to programming languages. Your “computer science book wants to explain the basic functioning of computers. This starts with the hardware and goes on to general computing concepts. “Python” is a wonderful programming language, but it is further removed from hardware than the C programming language. I will therefore also refer to this “mother tongue” of computer science at one point or another, the details of which you will find in the fifth part of this book! There are also some references to Java (Part VI). Part IV contains a general introduction to programming languages.

Python and other snakes

Before you can start programming in Python, you will of course first need a useful development environment. Fortunately, there are many of them. All important information (and downloads) about Python can be found here:

https: //www. python. org/

However, because there are so many libraries, also known as packages, for Python, and because you will probably want to run different applications in parallel over time, I recommend that you start with a Python distribution that contains a package management system. Here is another snake to consider: Anaconda.

https://www.anaconda.com/products/distribution

According to its own information, the open source version of Anaconda (there are also several commercial versions) is the world's most popular Python distribution.

Anaconda is available for all common operating system platforms and is easy to install, but depending on your computer equipment it may take a while. After starting the Anaconda navigator, you are again spoiled for choice (see Figure 29.1)

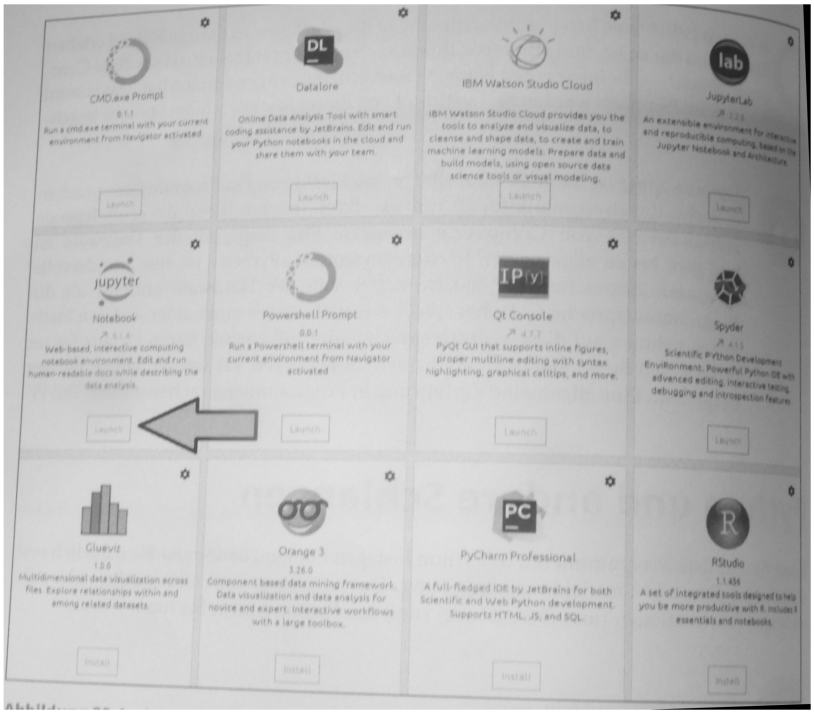


Figure 29.1 Selection of Python environments

I have marked the “Jupyter Notebook” with the arrow, which allows you to run Python interactively in the browser.

With pure compiler languages such as C or C++, you can only run your program after the complete source code has been compiled. The result is an executable file.

In contrast, Python is delivered with an interpreter where you can see what happens after each line of your program. This has great advantages when creating and debugging your code, but is not as fast in execution, especially for very time-consuming requirements. This is why there are also compilers for Python.

In chapter 17 you will learn in detail about the difference between interpreters and compilers!

Python can also be used as a scripting language. I explain what this is all about in chapter 48.

Snake worlds

It seems obvious to call a distribution of Python simply Anaconda. Both are types of giant snakes. However, according to its original inventor, the Dutch developer Guido van Rossum, the name of the Python programming language does not refer to the animal at all. According to van Rossum himself, he was a fan of the British “Monty Python's Flying Circus”. This also explains the correct pronunciation of “Python”, although this is certainly not a criterion for the usefulness of a programming language.

Although the origins of this language date back to the 1990s, it only achieved its breakthrough with version 3 from 2008, which is still being constantly expanded today. In this book, we only refer to version 3.

If you prefer, you can of course also work with Python in a “real” development environment such as “Spyder”. Or you can simply use the console - whatever suits you best. The main thing is that you end up with an executable version that you enjoy!

Everything installed, everything ready? Then you can finally get started!

Python basics

If we look at any C program, we immediately notice that the number of brackets, especially the round and curly set brackets, is enormous. There are also a huge number of semicolons. All variables must be declared

and defined, i.e. provided with a type and a value. All this makes an average C code quite confusing. However, these syntactic elements are also the most important structural feature. In addition, it is the handling of pointers in C that causes the biggest headaches and is also the most dangerous source of errors

You can find out everything you need to know about C in Part V of your computer science book in Chapter 21. You will find nasty tricks with pointers in Chapter 23.

However, you can format the source code as you wish and thus - in the worst case - make it much more unreadable.

There are programs that were written solely to reformat the source code of any C program to make it a little more readable. This is called “pretty print”, the embellishment of the source code without changing its meaning.

What if you completely dispense with all of these things and simply specify the structure by formatting the source code? Then you are welcome to Python! Start with a somewhat more exciting example, because the typical “Hello world”. program is really, really boring in Python:

print(“Hello dummies world!”)

No inclusion of any libraries, no preprocessor statements and no... Semicolon at the end of line n (like in “C”)!

Our first (real) Python program

Because Python is a language that can be learned quite quickly, we can start with a somewhat larger example right at the beginning. How about listing all prime numbers under 100?

```
def prim(zahl):
    if zahl == 1:
        return False

    if zahl == 2:

       return True

    for i in range(2,zahl):

        if zahl % i == 0:

            return False

        return True

for zahl in range(1,100):
      if prim(zahl):
            print(zahl)
```

If you already have programming experience with another language, you can breathe a sigh of relief here! The code is very concise and without embellishments (that would be curly brackets around blocks or round brackets after the keywords). You will also have noticed that the variables are used immediately without specifying an explicit type.

However, perhaps you are new to programming and have just chosen Python as your first language? Great, that's also a very good idea! Let's just go through the program line by line.

def prim(zahl):

This statement defines a new function to be called “prim”. The def is a keyword in Python and means that a new function is to be defined afterwards. You are free to choose the identifier prim; well, almost, you cannot use a keyword, for example, and the identifier should not start with a number either. Instead, you can use German umlauts, for example, to your heart's content - if you do not want to send your work abroad, where this might be a problem if the recipients cannot find the character on their respective keyboards ...

The parameter list of the function is enclosed in round brackets. Round brackets all of a sudden? Yes, that's right, we can't do without a few. But the zahl passed doesn't seem to have a type, at least you don't have to specify it separately. What do you have a computer for? It should find out for itself which type is best suited for this variable! This is called “implicit typing”.

Different types

Most programming languages require explicit typing of their variables. In C/C++ or Java, for example, you must explicitly specify whether a variable should be of type int or float.

Static typing also applies in C/C+4 and Java. This means that every variable can change its value, but the type is unchangeable. This means that it cannot change during the runtime of the program.

This is much more convenient in Python. Implicit typing means that the type of the variable should be inferred from the context. For example, if the assignment is zahl = 7, the variable zahl is assigned the type of an integer. With zahl = “seven”, on the other hand, you get a string. Dynamic typing also applies in Python: if you later assign a different type to a variable, this is accepted without question.

That sounds almost too good to be true. In fact, it works great, but there are also limits. If Python no longer knows (and cannot infer from the context) which type should be used for a variable, a TypeError is thrown, for example here

```
a = 3 + "four"
```

Should this be an integer or a string? Computer scientists call these types incompatible.

In contrast, other mixtures work very well: `a = 3.3 + 4`

A floating point number plus an integer can easily be added together, the result would be the floating point number 7.3. These two data types are therefore compatible.

In our large example program, Python knows after the line

```
def prim(zahl):
```

Python does not actually know which type is meant by number. This must be determined automatically later when the program is called. We will come back to this in a moment,

What is the colon at the end of the line for? Didn't we want to do without all the flourishes?

Yes, we do, and in fact you could imagine Python omitting this colon (at some point). But it has a very practical use:

After every line that ends with a colon, a block begins, which in Python is marked by an indentation!

What you do in C or Java only as a cosmetic representation of your source code, without any meaning, is crucial for Python. This solves three problems at once:

- You save the curly braces.
- You get nicely displayed code.
- You avoid the risk of misinterpreting a program because the formatting of the source code tempts you to do so.

You will understand the third tick when you read about the common errors that can occur in C due to forgotten curly braces in Chapter 22.

If your editor knows that you are programming in Python, it can simply - automatically - indent all subsequent lines by one tab step after a line ending with ":". This is very practical, and you will learn to appreciate it ...

Back to the example. There are further indentations in the body of the prim function:

```
if zahl = 1:
        return False
if zahl = 2:
        return True
```

First, the value of the variable number is compared with the if statement to 1. If this is true - and only then - the Boolean value (`False`) is returned (this is what the keyword return stands for). Note that the if query also ends with a colon and the following block (which in this case only consists of one line) is indented accordingly. In addition, you must write False in exactly the same way and not completely in lower case, as Python is a bitch in this respect (just like C). Computer scientists call this case-sensitive.

The third line begins with if again, but it must not be in the same block as the line above it, but must be indented "at the same level" as the first if.

In the case that number assumes the value 2, `True` is returned as the result of `prim`.

You may be wondering how to undo an indentation with the **Tab** (Tab ->| and Tab |<-). In many development environments (unfortunately not all) it works like this: **Shift + Tab**. It makes the cursor jump to the left. Both **Tab** and **Shift + Tab** may be used several times in succession. Python doesn't care how many spaces you indent, as long as you do it consistently ...

In other words: The prim function receives a number and first checks whether it has the value 1. In this case, it returns `False`. If, on the other hand, the value of number is 2, it returns `True`, because 2, unlike 1, is a prime number. And what if the number is even greater? The following code can be found in the function:

```
for i in range(2,zahl):
        if number % i == 0:
                return False
return True
```

The for-in loop is used to execute the following block (yes, again a colon at the end of the line) several times in succession.

Between `for` and `in` there is again a variable (or a tuple, we will come to this later) and between in and the colon there is a systematic collection of data.

In our example, this is a `range`.

Range

The range data type has up to three parameters: `range ( start, stop, step)`

With start you determine the smallest value of the number range, with stop the end (but exclusive, so this value no longer belongs to the range) with step the step size.

If you only specify two values, the step size is omitted; if there is only one element, 0 is simply assumed to be the start value and the remaining value describes the (excluded) end. The increment and all other W values may of course also be negative. So deliver one after the other:

- `range (100)`: all integers from 0 to 99
- `range (-10,10)`: all integers from -10 to 9
- `range (0, 51, 5)`: the series of 5 from 0 to 50
- `range (20, 0, -2)`: all even numbers from 20 backwards to 2 (the 0 as the last element is therefore also excluded when counting backwards).

At first glance, this may seem annoying: Why can't Python simply offer a `for` loop in which the start, end and increment are specified directly instead of using a new data type such as range?

Believe me: it's better this way! Because the range specification is just a special case. Take a look and see for yourself:

```
for b in "Hallo":
    print(b)
```

Instead of the range, just specify a string (in double quotes) and all the letters will be printed in order!

Or here :

```
for e in ["Liste", "mit", "Text", "und" , "Zahlen" ,2,3,5,7,0;5]:
    print(e)
```

A list is extremely useful, you can put together any other data types in it - separated by commas. Square brackets are placed around a list “[...]”. You will find all the elements of the list in sequence as output. The so-called loop variable in this example is `e`, you can of course choose this identifier freely. Lists are the most common and most important data type in Python, and we will take a closer look at them in this chapter!

But first back to the example: In our first real Python program, `range (2, zahl)` stands for the listing of all numbers from 2 to number-1.

If the end value of a range is less than or equal to the start value, this is not an error. The range is then simply empty and does not contain any numbers.

The inside (the body) of this for-in loop is not exactly spectacular either:

```
if zahl % i == 0:
```

First, a query takes place here as to whether "zahl modulo i" is zero. This is the computer version of "Is the number divisible by i?

The modulo operator in a % b specifies the remainder of the integer division of a by b. For example, 17 % 5 results in the value 2, because the remainder of the division of 17 by 5 is 2.

If this is the case, `False` is returned. In this case, the number passed is divisible by another number and is therefore not `prime`. If the loop does not find a divisor of the number, `True` is returned at the end.

Note that the line

```
    return True
```

is indented just as far as the for-in loop above it. It is therefore only executed after the loop has been completed! In other words: The function `prim` returns "`true`" if the number passed is a prime number.

In addition to the modulo operator, Python also has all the other obvious mathematical operators.

Mathematical operators in Python - overview

1. addition: +

2. subtraction:-

3. multiplication:*

4. division (fractional result): /

5. integer division (without remainder): //

6. modulo operator (remainder after integer division): %

7. exponentiation:**

If you use several operators in an expression, the “dot before dash rule” applies and the exponentiation binds most strongly.

You do not need brackets for a polynomial:

```
x**2 + 3 x -1
```

If x previously had the value 2, then the polynomial $x^2 + 3x - 1$ has the value 9

The priority rules of the mathematical operators are designed to be able to specify polynomials without brackets.

Now consider the “main program”:

```
for zahl in range(1,100):

        if prim(zahl):

                print(zahl)
```

The good thing is that you don't have to create a `Main` or follow any other strange conventions. The main program can simply be recognized by the fact that it is not indented.

In our case it is - once again - a for-in loop. The number range here is from 1 to 99.

In the loop body, you simply ask whether the current number is `prime`, if so, it is output with `print`. That's it. The program thus enumerates all prime numbers under 100.

Of course, our prime number program is not particularly efficient, because it would not have to examine all divisors, but only go up to the root of `zahl`. Furthermore, there are much better - and more complicated - methods for examining the divisibility of numbers. However, this only becomes important when we are dealing with very large numbers and goes deep into the mathematical field of number theory.

If you have made it this far, you deserve some praise: our first Python program was not that easy to understand! But that was less due to the programming language than to the task of enumerating prime numbers.

The most important data types in Python

Python is a very easy language to learn. In addition to the integers (int), which we have already used in the prime number program, there is a whole range of other types. I have listed the most important ones in the box:

Fixed data types in Python - Overview

1. number types

- `int` Integer of any size

- `float` Floating point number

- `complex`: complex number with real part and imaginary part

2. sequence types

- `list` Collection of any number of different values, which may themselves be lists or other data types. Lists are framed by square brackets [and].

- `tuple` Compilation of other data types into a tuple. The difference to a list is that a tuple, once created, must remain immutable. Therefore, all methods with which you would change the tuple are omitted. Tuples are usually framed by round brackets (and). However, you can omit these in most cases.

- `set`: Structure that represents a set. You have no control over the order in which your set elements are arranged. As with real (mathematical) sets, an element cannot occur more than once. Typical set operations such as union and intersection are available to you for this purpose. Sets are framed by curly brackets {and }. Unlike a set, a frozenset is immutable.

- `dict` dictionary {dictionary), also called a map or associative array in other languages. You do not have to index a dictionary with a numerical value, but can simply work with strings as indices, for example.

- `range`: Any range with start value, end value and increment

3. string

- `str` String, any character string

4. bool

- `bool`: Truth values True and False

5. other

- `None` the data type for ... nothing at all!

- `bytes, bytearray, memoryview` Data types for binary operation

With a few playful examples, I would like to introduce you to the essential aspect data types.

Lists

As already mentioned, lists are the central data structure in Python, and you can do a lot with them! In other languages, the list would perhaps be called a dynamic array.

```
my_list = [1,2,3, “today”, “is”, “a”, “nicer”, “day”]

print(my_list)
```

The output is:

```
[1, 2, 3, 'today', 'is', 'a', 'nicer', 'day']
```

You access an entry (for example) via the index. For example, `my_list[3]` returns the value 'today', which means that the count always starts at 0. You can also see that the output string (character string) 'today' only has single quotation marks, whereas we had specified “today” with double quotation marks in our list. Python is very relaxed about this. You can do it either way, just, of course, consistently: If you start the string with a single apostrophe, you must also end it with it - and vice versa.

With `my_list [3] = 'tomorrow'` you replace “today” with “tomorrow” in the list,

You can use append to add an element to lists:

```
new_list = []

for i in range(10):

        new_list.append(i)

print(new_list)
```

The result [0, 1, 2, 3,4, 5, 6,7, 8, 9] could have been simpler:

```
new_list = list(range(10))
```

As you can see, the keyword `list` creates a list from the given enumerable, iterable object.

Did you notice that I chose “`my_list`” instead of “`mylist`” as the identifier?

The first version is considered pythonic, which is also called snake-case and differs from camelCase. Can you recognize the **T**wo **H**umps in the camelCase style? Our Python snake, on the other hand, prefers lower_case_with_underscores because it supposedly makes it easier to read. This is a convention that you do not necessarily have to adhere to, the interpreter also processes the camelCase without any problems.

In addition to `append()`, there are many other ways to change the list. For example, `extend()` would append another list (element by element). You want to sort the list? No problem! As long as the elements are of the same type (e.g. strings of numbers), this is very easy. With `new_list .sort()` the list is transferred to a sorted state, with `sorted(new_list)` you get a (new) sorted list as a return, but your original (`new_list`) remains unchanged.

Syntactically, Python adopts the convention of appending method calls to the respective object using the dot operator ".". A more detailed discussion of object-oriented programming with Python can be found in the next chapter.

How are you supposed to remember all this? Fortunately, Python has a built-in help function, `help ()`, which you can use for all kinds of built-in (and custom!) data types. Among other things, you will find the useful method `count ()`, which counts the occurrences of the respective entries in the list.

You can also use `count()` directly on strings.

For example, the line `"Hello world".count(" l")`

produces the result 3, because there are three "l "s in the string ...

Help function in Python

With `help` you get a quick overview of the entire range of Python functions!

Try out `help(list)` or the help function for another data structure or built-in function yourself!

Your own functions can also be explained using `help()`. All you have to do is fill in the docstring after the declaration. It works like this:

```
def my_function():
        " "" This is my function that does nothing! " ""
return None
```

`help(my_function)` now returns:

```
Help on function my_function in module_main_ :
my. function()
        This is my function that does nothing!
```

Tuple

The tuple is similar to a list. The elements may also be arbitrary, but they may not be changed afterwards. You can also omit the brackets if you wish:

```
my_first_tuple = (1, “a”, 0.5)

my_second_tuple = 1, “a”, 0.5
```

You will have a lot of fun with tuples if, for example, you want to assign more than one function value to a function: After the return you return (separated by commas) as many values as you want.

Thus

```
def many_return_values():

return 1, 2, “3”

a, b, c = many__return_values()

print(a,b,c)
```

as output:

```
1 2 3
```

This also makes it clear that you can also obtain the individual elements of a tuple (except by referencing them directly with []) by enumerating several variables on the left-hand side of the equals sign. Actually ingenious!

Sets (set)

Classic set operations are available to you with sets:

```
M = { “a”, “b”, “c” }

N = { “b”, “c”, “d” }

print("Union set: ”, M | N)

print("Intersection : ”, M & N)
```

results in:

```
Union set: {'c', 'a', ' b', 'd'}

Intersection: {'c', 'b'}
```

Instead of the short `M|N`, you could also have written `M.union(N)`, or `M.intersection(N)` instead of `M & N`. Simply choose what is more legible for you. Again, you will find a very large selection of set operations using help(set).

By the way, with `frozenset()` you create an unchangeable variant of a set. This corresponds roughly to the difference between a tuple and a list. In general, access methods can be implemented somewhat more efficiently for immutable objects.

Dictionaries (dict)

Dictionaries also leave nothing to be desired. You can define entries for dictionaries using pairs separated by colons. At the end, as with sets, curly brackets are placed around the final result. After the definition

```
German_english = { “Hund” : “dog”, “Katze”: “cat”, “Vogel” : “bird”}

German_english [“ dog” ] results in the string “ dog”,
```

Here too, the for-in loop is very suitable for listing the content of a dictionary:

for `key in German_englisch:`

```
        print(“The translation of”,German_english[key], “is”,key)
```

The result here is:

```
The translation of dog is Hund

The translation of cat is cat

The translation of bird is bird
```

You can solve the text output much more elegantly using the String-method **`format`**. But please be patient. This will come in the next chapter (30)

Of course, with dictionaries (just as with `Lists` and `Sets` or your own `Container_objects`) you also have the option...

- use `len(object)` to determine the number of entries.
- remove the last entry using `object. pop()`: Incidentally, this removes an arbitrary element from a set. For a dictionary, however, `pop()` requires an argument.
- `pop( index)` to remove the entry at the position “`index`”. In dictionaries, this index is also referred to as the key.
- simply read or change the entry at the position “`index`” with `object [index]` (using an assignment: `object [index] = new _entry).`
- As far as dictionaries are concerned, the last point is a little more complicated.
- Take another look at our example with the dictionary German_english. This means that adding entries is not critical:

```
German_english [“Fisch”] ="fish”
```

- If there is already a different translation for “fish”, this is simply overwritten. Conversely, this is somewhat more problematic:

```
print(German_english [“Kuh”])
```

In our example, this command returns a KeyError because the key “cow” does not exist. To find an elegant solution here, Python offers the `get()` method:

```
print(German_english.get(“Kuh”))
```

The method returns `None` if the entry does not exist (without an error message). It is now nice that you can also define a different return value.

```
print(German_english. get(“Kuh”, “Entry unfortunately does not exist!”))
```

This return value shows the user immediately what is going on. If the “Kuh” does exist (in the meantime), the second parameter is ignored.

In the next chapter, I will come back to the control and data structures and show you what else you can do with them.

Chapter 30 The colorful wide world of Python

First of all, this chapter deals with the concept of “comprehension”. It can be applied to lists, sets or other types of enumerable data types (“iterables”). It is therefore recommended that you first study the basics from the previous chapter.

I will then introduce you to the possibilities of string processing in Python. There is nothing left to be desired, so look forward to it. I will also show you how to add any kind of parameters to your own functions.

Finally, we will look at the implementation of object-oriented concepts in Python. There, too, attention is paid to a syntax that is as simple and easy to learn as possible, but at the same time very powerful.

Comprehensive understanding

The word “comprehension” means “understanding” or “insight”.

In English lessons, you have probably been confronted with “reading comprehension” or “listening comprehension”.

In Python, the concept of comprehension refers to the construction of lists, sets or other enumerable data types as elegantly as possible.

The idea is borrowed from set theory in mathematics. You probably remember the following notation for sets from your school lessons:

$M = \{ n^2 \mid 1 \leq n \leq 15\}$

Python now translates this approach - as closely as possible - into the following program syntax

```
M = [n**2 for n in range ( 1 , 15 ) ]
```

Instead of the square brackets (which generate a list), you could also have used the brackets to generate a set. Either way, M then contains all square numbers between 1 and 196 (which is the value of 142):

```
[1, 4, 9, 16, 25, 36, 49, 64 , 81, 100, 121, 144, 169, 196]
```

Still not convinced? Let's do a slightly more complicated example;

$N - \{(x,y) \mid 1 \leq x < 7 , 1 \leq y < 7 , x + y = 7 \}$

The set N therefore contains 2 types, each from the number range between 1 and 6, but the sum of both values must be exactly 7.

In Python it looks like this:

```
N = [(x,y) for x in range (1,7) for y in range (1,7) if x + y = 7 ]
```

For example, you can consider the result as the set of all double dice (with normal dice) where the sum of the eyes is 7.

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

Impressed? You can nest the whole thing even further and represent really complex sets immediately.

Do you remember the “sieve of Erastosthenes”? This can be used to construct sets of prime numbers, for example those under max_prim

```
max_prim = 20

Prime numbers = { p for p in range (2, max_prim) if p not in [ n for m in range ( 2,max_prim//2 ) for n in range ( 2*m, max_prim, m) ] }
```

This is quite complicated. The outer set-comprehension enumerates all the numbers (p) from the number range from 2 to max_prim. The restriction after that, if p not in, refers to the following list, which is also generated using a comprehension: All n are enumerated that range from 2m to max_prim, where m itself again comes from the range 2 to maxprim//2 (integer division by 2).

What does this mean? In the nested list, the series of 2s, series of 3s, series of 4s and so on up to the max_prim//2 series are listed one after the other.

And the restriction in front of it requires that p does not occur in this list. In other words: The (outer) set contains all elements that do not occur in one of the number sequences and therefore represent prime numbers. The result in the example (with max_prim = 20) is:

```
{ 2, 3, 5, 7, 11, 13, 17, 19 }
```

This (not very efficient) way of constructing prime numbers has gone down in the history of mathematics under the term “sieve of Erastothenes”.

Putting characters in chains

Every programming language has to solve the tiresome issue of strings. Texts are constantly being output and should still look properly formatted. In addition, all other possible data types must also be converted to strings. Last but not least, you want to search in strings, perhaps split them, re-sort them or count letters in them. These are all reasonable requirements that Python is also confronted with. And what can I say? They are solved with flying colors!

The data type used in Python is called `str` and has all the necessary operations, see for yourself:

```
s = “This is a string, we enjoy it !”

s.split()
```

The result of the split:

```
['This', 'is', 'a', 'string, ', 'we', 'enjoy', 'it ! ' ]
```

Maybe you don't want to split s by spaces (and other whitespaces), but by the comma? No problem, the separator is simply passed as a parameter:

```
s.split (” , ”)

['This is a string', 'we enjoy it !' ]
```

Are the first three words enough for you? Here you go:

```
s.split (None, 3)

['This', 'is', 'a' , 'string, we enjoy it !']
```

The None as the first parameter is important because otherwise the 3 would take on the role of the separator string, which would lead to an error: Split expects a string as separator in any case - or None, then it's the whitespaces again...

`rsplit` you split the string from right to left, which only makes a difference if you add the number.

`splitlines` are used to split the linefeed-separated parts of a long string, which is often used when processing “longer texts”, for example from a file

`partition`, you create a tuple consisting of the left-hand part of the string, the separator itself and the part to the right of the (first) occurrence. If the separator string does not occur at all, the two tuple elements at the back remain empty. (Chapter 35 contains a concrete application of the partition in the `quicksort` algorithm).

`rpartition` works like partition, but searches for the first occurrence of the separator from right to left instead of vice versa.

`join` allows you to provide a list of strings with the same element in between. This works in exactly the opposite way to split

If you want to search for a character string in a string, the easiest way to do this is with `in`.

```
“Spass” in s
```

results in true. If you want the exact position, use `find`

```
s.find(“Spass”)
```

also returns the same result if successful, but otherwise throws an exception. Therefore, it must not be used in the condition part of an instruction and with `rindex` you search from the right again.

Do you want to count substrings in a given string? You can do this with count

```
s.count(“s”)
```

returns a 3 in our case, because the lowercase “s” appears three times. This brings us directly to the question of how to convert a given string into upper or lower case letters:

```
s.upper()
```

```
'THIS IS A STRING, WE ENJOY IT!
```

```
s.lower()
```

```
'this is a string, we enjoy it! '
```

In addition, there are a myriad of methods to determine whether a string consists of characters, digits, all uppercase letters and so on. Do some research under `isalphaQ, isdigitQ, isalnumQ, isupperQ, islower{), isspaceQ...`

If you want to combine strings with a print output, this is very easy, as we have already seen. For example leads

```
inte_number = 23

float_number = 47.1199

String = “Hello dummies world”

print (“Numbers:”, inte_number, “and”, float_number, “and”, string)
```

to

```
Numbers: 23 and 47,1199 and Hello Dummies World
```

The story becomes even nicer if you use format():

```
print ("Numbers: {0} and {1} and {2}”. format(inte_number, float_number, string)).
```

As you might expect, the numbers in the curly brackets correspond to the respective parameter of the attached `format()` statement. As always, counting starts at 0.

If the order of the {} brackets corresponds to those in `format( )` and each one is only to be used once, you can omit them completely:

```
Print ("Numbers: {} and {} and {}”.format(inte_number, float.number, string))
```

Do you prefer characters instead of numbers? No problem:

```
Print ("Numbers: {z} and {f} and {s}”. format(z=inte_number, f=float_number, s=string))
```

With a preceding “f” you can also make a string to a format string outside of print, which interprets variables outside, like this:

```
number = 30

word = “Dummmies”

string = f“{number} is a nice number in your {word} book!”.
```

Then string has the value:

```
30 is a nice number in your Dummies book!
```

Speaking of special strings: In addition to the preceding “f”, there is also an “r” for “raw”. Such a raw string does not interpret its special characters inside. This is very useful, for example for path specifications:

```
r “C: \Users\haffner\Desktop\Python ”
```

If you were to omit the “r”, Python would interpret the various backslashes as the start of special characters and produce a nasty error message in this particular case:

```
 SyntaxError: (unicode error) ’unicodeescape’ codec can't decode bytes in position 2-3: truncated
\UXXXXXXXX escape
```

As a raw string, however, all characters are simply displayed as they were entered!

The formatted output of floating point numbers corresponds to the model in C:

Sparking functions

In my humble opinion, the syntactic strength of Pythom is particularly evident when dealing with functions. They behave exactly as one could wish. No complicated thinking about pointers, handles or memory structures. Just get started:

```
def double(number):
        “”” double number “”
        return number * 2
```

This function multiplies the argument by 2. By the way, the line with the triple quotes is called “docstring”, which was already mentioned in the previous chapter, I'll come back to it in a moment.

It is not surprising that the instructions

```
double (14)
```

```
or also
```

```
a = 14
```

```
double(a)
```

both show 28 as the result. You can also assign the function to a variable like everything else:

```
f = f( 17)
```

You will be surprised that our f (and of course the original `double`) can also accept lists as parameters:

```
a = [1,2,3,4]
```

```
f(a)
```

Leads to

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

Multiplication by 2 (of a list) doubles it in another sense: The entries are all listed twice. This is something you can also do with the list without a function. What is exciting, however, is that our function has not decided on a specific type, but only decides at execution time what exactly is to be done.

If you now use

```
help( double)
```

appears:

```
Help on function double in module_main_:

double (number)

        “”” double number “”
```

There it is again, our “docstring”. After reading the next section, you will also understand that you can also call this string directly with

```
double._doc_
```

directly...

Do you need default values in your arguments? Python also provides a very simple scheme for this:

```
def function_with_default_values(number1 = 1, number2 = 2):

        return number*number2
```

You can simply use “=” to specify the respective value. This is only used if no other value has been passed. These are also referred to as key parameters. The word number1, for example, would be the key to the first parameter.

This results in...

```
function_with_default_values()

function_with_default_values (number 2 = 4)

function_with_default_values (number1 = 5)

function_with_default_values (number1 = 3, number2 = 5)
```

successively the numbers 2,4,10 and 15.

If you do not know how many parameters are entered, you can use an asterisk:

```
def many.parameters (*many):

        return many
```

The return results in a tuple.

What was that again about tuples? You can find an example of the interaction between tuples and functions in chapter 29

`many_parameters (1, 2, “Hello”)` leads to

```
(1, 2, 'Hello')
```

A tuple that should only contain one value, for example 123, must be defined with = (123,). Admittedly, this looks very strange: A comma but nothing after it? Unfortunately, it is necessary. If you omit the comma, z = (123) simply results in the number 123 and not a tuple with just this one value...

An arbitrary (*variadic*) number of key parameters is also possible:

```
def any_key_parameter(**arguments):
        for key in arguments:
            print(“key =”, key, “and val =” , arguments[key])
```

The two asterisks in succession turn the argument into a dictionary that you can then “decrypt” again - as *key-value* pairs.

```
any_key_parameter ( a=12, b=10, c=” dummies ” )
```

results in

```
key = a and val =12
key = b and val = 10
key = c and val = dummies
```

This leaves nothing to be desired ...

Of course, functions can also call themselves. The concept is called *recursion*.

We take a look behind the scenes of recursion in general - not just Python - in chapter 33 (from the practical side) and chapter 54 (from the theoretical side).

The classic says it all:

```
def faculty(n =1):
        if n <= 1:
                return 1
        return n* faculty(n-1)
```

The function calls itself in exactly the same way as it would with another function. Just don't forget that something must always become smaller during the recursive call. Otherwise you will produce an infinite loop which will very soon lead to an error:

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Exceptions in Python

Exception handling in Python follows the familiar “throw-catch” scheme, which is done syntactically here with try and except

Chapter 18 introduces you to the problem of exception handling...

A typical example is the “division by zero problem

```
numerator = 1

denominator = 0

try:

        print(numerator / denominator)

except :

        print(“Division by zero forbidden!”)
```

The *try* block can contain one or more critical commands, even in sub-calls. As soon as the problem occurs, the rest of the code is exited and the *except* block is executed immediately afterwards. The latter is skipped if no exceptions have occurred.

Following the *except* block, an *else* block can - but does not have to - be added, which is only called if no error has occurred.

If you then want to ***always*** execute further code, use the (again optional) *finally* block!

You can also append several except blocks to each other if you want to differentiate between the various exceptions. For example, ...

```
except ZeroDivisionError as e:

        print(“Division by zero forbidden!”)
```

would only handle our error above. The `e` contains the error description itself, which is assigned via “`as`”.

Conversely, you may be wondering how to throw an error? You can do this with `raise`

Here is another example that contains everything once again and that you can try out for yourself as you wish. A keyboard input is accepted with input. The attempt to interpret this as an integer using Int is already problematic and can provoke various exceptions.

```
def fractionalvalue():
        result = None
        try:
                Numerator = int(input("Numerator:"))
                Denominator = int(input("Denominator:"))
                result = Numerator / Denominator
        except ValueError as e:
                prlnt("Error:",e)
        except ZeroDivisionError as e:
                print("Division by zero")
        except:
                print("other error, will continue*)
                raise
        else:
                print("No error occurred")
        finally:
                print("finally is always executed")
        return result
```

You can produce the errors yourself in this small program. For example, enter a 0 for the denominator or a letter instead of a number. After that, everything should be `clear`!

Generators and factory functions

In the introduction to the `for` loop, you have already learned about `range` as a way of creating an ***iterable*** object. Perhaps you would like to program such enumerable structures yourself? The concept of ***generators*** was introduced in Python for this purpose. A generator is a function that produces an "***iterable"*** This is essentially done by using the keyword `yield` instead of return

As an example, I will show you a simplified version of ***range*** as a generator:

```
def simple_range(stop):
        current = 0
        while current < stop:
                yield current
                current += 1
```

From now on you can use `simple_range` like `range` (with only one parameter)! If this is a little too fast for you, you can also simply call this generator step by step. It works like this:

```
my_ generator = simple_range(5)
print (next(my_generator))
print (next(my_generator))
print (next(my_generator))
```

Each call to ***next*** executes the generator code up to the next yield - and stops immediately afterwards! This continues until the code has actually reached the end. Then a ***StopIteration*** exception is thrown, which you can of course explicitly intercept...

The generators are particularly useful in situations where you potentially produce a large amount of data, but don't know how much from the outset. Imagine that each call to the generator (until the next `yield`) would be very time-consuming. Perhaps a `break` occurs within a `for` loop that uses the generator. Then only as much data would actually be generated as was needed, even if the generator had been called with 1,000,000 as an argument!

I explained above that the syntactically unconstrained handling of functions is a particular strength of Python. This is especially true for "factory functions", functions that you can produce (like in a factory) with every call!

Let's take linear functions of the form **f(x) = a-x + b** as an example. For each of the two coefficients **a** and **b**, a different function **f** results.

```
def linear_function(a,b):
        def f(x):
              return a*x + b
        return f
```

The inner function **f** uses the parameters **a** and **b** of the outer function! The best thing is that `linear_function` simply specifies the inner function f as the return value. That's it! Each call to `linear_function` now creates a (different) function **f**. Let's try this out:

```
f1 = linear_function(2,3)
f2 = linear_function(3,4)
print(f1(3))
print(f2(-1))
```

`f1` corresponds to the linear function 2x + 3. Therefore, calling `f1 (3)` results in the value 9, since 2*3+3 = 9. `f2`, on the other hand, represents the function 3*x+4. `f2(-1)` therefore results in 1, since 3*(-l)+4 = 1.

Oh dear - now it's all about OOP

Of course, Python also implements the paradigms of object-oriented programming (OOP) - like any modern language that is worth its salt.

A general introduction to OOP can be found in Chapter 17. The implementation of OOP in C++ is the topic of Chapter 24.

Let's start with a small example.

```
class Tree:
       def_init (self, name, alter):
          self_name = name * public property
          self_ age= age * private property
         def output(self):
          print("I am a {1} year old {0}." .
               format (self. name, self._age))
         def grow (self):
               self._age += 1
```

The "Tree" class has the three methods `_init_(), output()` und `grow ()`.

The reference to the current object is `self` and is always the first parameter.

Strictly speaking, self is only a convention. The first parameter always refers to the current object instance, no matter what you call it. However, you should stick to this convention to make the codes easier to read. In C++ and Java, the object instance must also be referenced with ***this***.

The underscores `(_)` at `_init_()` and `_age` are important!

Underscores in Python

The underscore "_" has a special meaning in Python, depending on the context it stands for different things:

- Outside of classes, a simple leading underscore helps you avoid a name conflict with an existing variable (or even a keyword).
- Within the class declaration, public attributes should never begin with an underscore. If you want to avoid a name collision here, use a single underscore at the end of the identifier (trailing).
- If you do not want an attribute to be available as part of the inheritance of a derived class, you should use double underscores in front (but not at the end).

- Within a class, simple underscores in front indicate that the corresponding attributes should only be used internally (not publicly). However, this is not strictly enforced (and is therefore sometimes referred to as “weak private”).
- Double leading and trailing underscores stand for internal methods in classes. _ init_is a good example of this
- The underscore also serves as a placeholder for a nameless variable. For example, if f returns 3 values, but you only need the second one, you can specify the other two with underscores:

```
_,a, _= f()
```

This always works, inside or outside a class

- And a little fun aside: if you simply enter the underscore as a command in the command line of the Python interpreter, the last output simply appears again!

You can then instantiate objects and execute methods:

```
b = Tree( “oak” ,120)
b.output()
```

results in:

```
I am a 120 year old oak tree.
```

And then

```
b.name
```

leads to

```
'oak'
```

While

```
try:
      print(b._age)
except:
      print(“Age is private!”)
```

results in:

```
Age is private!
```

There are two different ways in Python to output any object as a string.

- `_repr_` is intended to represent a unique representation of the object, with which the object can theoretically be created.
- `_str_` on the other hand, should simply provide a human-readable form of the object.

The best way to show you this is with a simple example:

```
Class Output ()
      def _repr_ (self):
            return “Output using _repr_”
      def _str_ (self):
            return “Output using _str ”
```

Now you get for:

```
a= output( )
```

with the input of

a

in the command line as the result:

`Output` using `_repr_`

while

```
print(a)
```

results in

`Output` using `_str_`

Alternatively, you could have obtained the respective representations using str(a) or repr(a)!

As long as you do not overwrite these methods, you will receive the default representation.

This is how the object `b` from our penultimate example results:

```
<_main_ .tree object at 0xl 0x000001D4361a8730>
```

You will of course see a different memory address. The string representation again falls back (more or less) to `_repr_`

```
print(b)
< _main_ .tree object at 0x00000104361A8730>
```

You should always think about the respective representation for your own objects.

For our tree class, you could simply choose the output as a string representation,

while `_repr_` should look like this:

```
def _repr_ (self):
return f “Tree(‘{self.name)’, {self._age})”
```

Incidentally, this is a very natural example of a format string with a preceding “f”, which you already learned about in the second section of this chapter!

For the tree instance b from the previous example, you get:

```
Tree ('spruce', 121)
```

This representation would therefore be suitable for creating the object (again). You can actually do this by using the `eval` function:

```
b2 = eval(repr(b))
```

produces a new instance b2 of the Tree with exactly the same properties as b.

You can use the `eval` function to execute a string as Python code in the context of the current variable assignment. The same works for the result of the compile function, which requires a Python source file as a parameter.

This is a good time to talk about Python objects in general. All classes inherit from the top-level root class object.

In addition to the string representations, object has numerous other properties that are automatically inherited and that you will therefore automatically find in all your own classes.

These include methods for comparison such as `_eq_` , `_ne_` , `_ge_` , `_ie_` , `_gt_` and `_lt_`, which you must of course overwrite in order for this to make sense.

The meaning of the abbreviations can be easily understood using the English comparison operators:

- `eq`: equal:, =
- `ne`: not equal:, #
- `ge`: greater than or equal:, ≥
- `le`: less than or equal:, ≤
- `gt`: greater than:, >
- `lt`: less than:, <

To determine the size in bytes, use `_sizeof_`

This is only a small excerpt. For the full scope, I recommend

```
help(object)
```

Of course, you could fill an entire book with the OOP concepts of Python alone. Simply put, it implements everything you would expect from a modern language! From introspection to multiple inheritance, nothing is left to be desired.

Instead - in the next chapter - take a look at the standard libraries that cover all possible fields of application.

Chapter 31 Luring Python out of its basket

It is extremely easy to integrate new libraries, which are called modules in Python, into your program code or to outsource your own programs in modules. How exactly this works is the first topic of this chapter.

There is already a huge, almost unmanageable variety of modules. As an example, I would like to show you the standard modules ***NumPy*** and ***Matplotlib*** and ***Pickle*** in more detail. Finally, let's take a look at what else is available beyond the edge of the plate...

Python is also jokingly referred to as “batteries included” due to the large number of included libraries!

Manage modules

If you want to include a library in your program code, this is done with `import`

```
Import numpy
```

This gives you automatic access to all functions of the numeric Python module. However, you must always prefix the module name:

```
Numpy.array([1,2,3])
```

Because this is a little tedious, you can also define an abbreviation when importing:

```
import numpy as np
```

From this moment on, the short form np replaces the long form:

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

Although you have a free choice of abbreviations, you should stick to common terms so that you can better understand other people's code and, conversely, so that your code is more easily understood by others.

If you do not need the entire module, but only certain functions, you can also specify a restrictive formulation:

```
from numpy import array
```

In this case, you must not specify the prefix when calling the respective function.

```
from numpy import *
```

However, I advise you not to do this because it can lead to name conflicts with your own methods.

As soon as you have imported a module, you can get started.

Do you need today's date? No problem:

```
from datetime import date
today = date.today ()
print(today)
```

There is a large number of modules that are automatically supplied with the installation of Python. They form the (*Standard Library*)

Comprehensive documentation of the Python standard library can be found at

```
https://docs.python.org/3/library/index.html
```

However, you will find a much larger number of available modules outside of these standard libraries on the net.

Python provides the package manager ***pip*** for this purpose.

pip is an abbreviation for “package installer for Python”

You can install an external module like this, for example:

```
pip install chess
```

In addition to the installation (`install`), you can list installed packages (`list`). Display information about packages (`show`), check packages (`check`) and much more. The (very comprehensive) concept of package management in Python is covered in the original documentation:

All info about the Python installation can be found at:

```
https,: //docs. python. org/3/installing/index. html
```

NumPy for home use

To give you a little taste of the power of the Python modules, I would like to show you what you can do with the NumPy module imported in the last section, especially with the NumPy arrays it contains.

Fancy indexing

Let's start with fancy indexing, the “fancy” selection of elements from a NumPy array. This allows you to use lists as a set of indices to select elements from arrays:

```
a = np.array([“I”, “N”, “F”, “0”, “R”, “M”, “A”, “T”, “I”, “K”])
index_list = [4,3,5,6,1,7,8,9]
a[index_list]
```

The NumPy array a here consists of letters, the usual Python list (which must consist of numbers) is now used as a set of indices from a to form a new word. And this is the result:

```
array(['R', '0', 'M', 'A', 'N', 'T', ' I', 'K'], dtype='<U1')
```

You can already see the special feature of the reference string: All elements of a NumPy array must be of the same type (unlike the Python list)!

Slicing

Slicing is the advanced selection of data in arrays. Incidentally, this also works with normal Python lists, but also with NumPy arrays.

First, we use arange to create an array of numbers directly as a NumPy array

```
a = np.arange(20) '-
print(a)
```

The result is:

```
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ]
```

This is quite convenient, you can also insert or specify initial values. If you only want to get certain parts of the array, you can do this with the aforementioned slicing:

```
a[2:8]
```

This creates an array from the number 2 (of course we count the indices from 0) up to the number 7. The end is therefore excluded as usual, while the beginning is included.

You can also omit the beginning or the end completely.

```
print(a[:5])
print(a[5:])
```

You get:

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

```
[ 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]
```

Theoretically, you can also omit the beginning and the end: `a[ : ]`, so you simply get a itself!

A step size as a third argument is also feasible:

```
print(a[2:12:3])
```

The result is:

```
[ 2, 5, 8, 11]
```

Only every 3rd element is output. If you want all even numbers from a, this is also no problem with `a[::2]`.

The increment may also be negative: `print(a[11:3:-2])`

You get the elements of a backwards, starting with the 11th element (inclusive) to the 3rd element (exclusive), each in steps of two...

```
[11 9 7 5]
```

This is even funnier with multidimensional arrays. First, we bend the (new) array a into a 5x6 array called multi using `reshape()`:

```
A = np.arange(30)
multi= a. reshape( [5,6] )
```

reshape() allows you to transform an array into a new form. This is passed as a list or tuple. Make sure that the product of the numbers corresponds to the number of elements in the original array, otherwise an error will be thrown!

This results in

```
array ( [ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23],
        [24, 25, 26, 27, 28, 29]])
```

you can use slicing to cut out parts of the array like a surgeon. With

```
multi [2:4, 1:4]
```

you get the third and fourth row and (in each case) the second to fourth column:

```
multi [2:4, 1:4]
array ([[13, 14, 15],
        [19, 20, 21]])
```

Again, you can also change the increments. If you want whole columns (or rows), slicing is even easier to achieve. With `multi [1:, ]` you get all rows from the second row onwards, while `multi [1, ]` simply results in the second row .

You can imagine that this goes on and on with even more dimensions...

That's not enough for you? You can also create entire grids of elements with the NumPy method `mgrid`, connect arrays with `concatenate`, split them with `split` (also horizontally `hsplit` or vertically `vsplit`) and add a dimension to your array with `newaxis`.

You can also accumulate or reduce NumPy arrays: multiply. `reduce` multiplies all numbers in the array and returns the product. With `add.reduce` you get the sum. If you use the accumulate method instead of reduce, the respective result is an array with all intermediate values. There are also lots of other mathematical functions. For example, continue your research under `help(np.multiply)!`

Of course, you can also determine the sum (`sum`), the minimum (`min`), maximum (`max`), the mean value (`mean`), the median (`median`) or many other properties of a NumPy array.

Of course, there is also `sort` to sort the contents of an array according to your taste!

Particularly worth mentioning are `argmax` and `argmin`, which this time do not return the maximum or minimum of the array, but the index of the respective element!

Vectorization

It's nice that you can do things with arrays in Python that would otherwise only be possible with numbers. For example

```
a = np.arange(5)
print(a)
print(a+3)
print(a/2 -6)
print(a+np.random.rand())
```

for output:

```
[0 1 2 3 4]
[3 4 5 6 7]
[-6. -5.5 -5. -4.5 -4. ]
[0.78219628 1.78219628 2.78219628 3.78219628 4.78219628]
```

Although ***a*** is an array, you may use mathematical operators as if it were a single number.

In the last command you can see that this also works if the operand is not a constant but a function - in this specific case (`random.rand`) a random number between 0 and 1.

Things get particularly exciting when you ***vectorize*** your own functions, which are actually only intended for individual scalars (simple numbers or other individual data types).

```
def my_letter(letter):
   return chr(ord(letter)+1)
```

The function expects a letter and outputs the next letter. For example, for

```
my_letter("A")
```

you get the letter "`B`" as the result.

If you now want to ***vectorize*** this function, it is very simple:

```
my_word = np. vectorize(my_letter)
```

to determine the result.

This function now accepts entire lists of letters:

```
my_word (list(“Dummies” ) )
```

you get `[ 'E' 'v' 'n 'n' 'J' ' f ' t' ]` (the following letters of “Dummies” as a result)

In this context, Python also speaks of “universal functions”, or “ufunctions” for short. You will understand the internal mechanisms of Python better after reading Chapter 32 if you extend Python with C code.

Masking without carnival

Another nice way to determine elements of an array is masking. To do this, you need to understand that you can use vectorization to produce boolean arrays. For example:

```
a=np.arrange(10)
a=5
```

The second line gives you the Boolean array:

```
Array ( [True, True, True, True, True, False, False, False, False, False])
```

You have probably already guessed that - similar to fancy indexing - you can use the result to mask the original array (or another of the same size):

```
a[a<5]
```

The result is an array with all elements of the required property:

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

Nice, isn't it? You can also use the usual Boolean operators. For example & for AND | for OR or ^ for XOR.

If your eyes light up now as you realize the power of the concept for your own applications, you have understood everything correctly!

Create graphics with Matplotlib

Is all this too dry for you? If you're interested in aesthetics, you've come to the right place!

A very widely used library for plotting mathematical graphics for Python is called Matplotlib. It is even more fun in combination with NumPy.

As a classic example, I will first show you how easy it is to plot the trigonometric functions of sine and cosine:

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

x = np. linspace(-np. pi, 3*np. pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```

First import **NumPy** and **Matplotlib**. For the latter, **Pyplot** is sufficient for the purposes of this small teaser; the usual abbreviation for this is **plt**.

We have not yet dealt with the NumPy function ***linspace***. This function simply produces equally distributed values in a given interval. In the example, it becomes a NumPy array whose first (and smallest) element is -π, and largest receives the value 3π. The third parameter specifies how many elements should be present in x in total: in our case 100.

The variables y1 and y2 then become fields with the corresponding cosine values. You are already familiar with the concept of vectorization, and this is a nice example of it.

The small but very powerful "***plot***" command then does all the work. So that the system recognizes when the result of the various plot commands is displayed, you will find the instruction at the end that the "show" can start!

The result (in Figure 31.1) is not bad at all for the very little effort involved

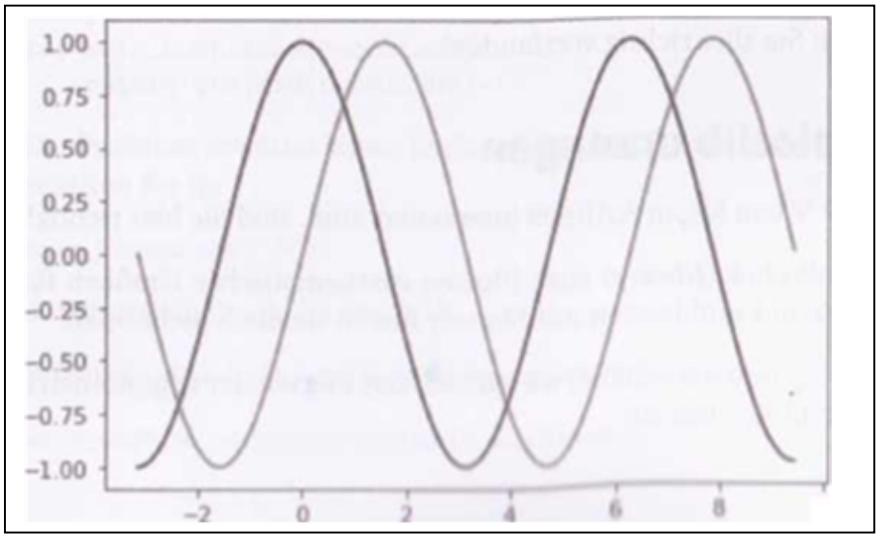


Figure 31.1: Sine and cosine plots with Matplotlib

You can change the look of the graphics to your heart's content using “style”. But don't complain if you spend many hours on this before you are finally satisfied: Unfortunately, this is always the case.

However, I would like to show you how to insert a legend, give the graphic a title and change the colors and line types:

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

y3 = x**2/90
plt.plot(x,yl, linestyle= “solid”, color="blue",label = "sin(x) ")
plt.plot(x,y2, linestyle= “dotted”, color = “green”, label = "cos(x)")
plt.plot(x,y3, linestyle = “dashed”, color="red ",label = "Parabola")
plt.title(“sine, cosine and parabola”)
plt.xlabel(“x”) plt.legend()

plt.show()
```

The function y3 is a parabola: $x^2/90$. I have inserted the denominator so that the parabola runs nicely across the image (Figure 31.2).

Sine, Cosine and parabola

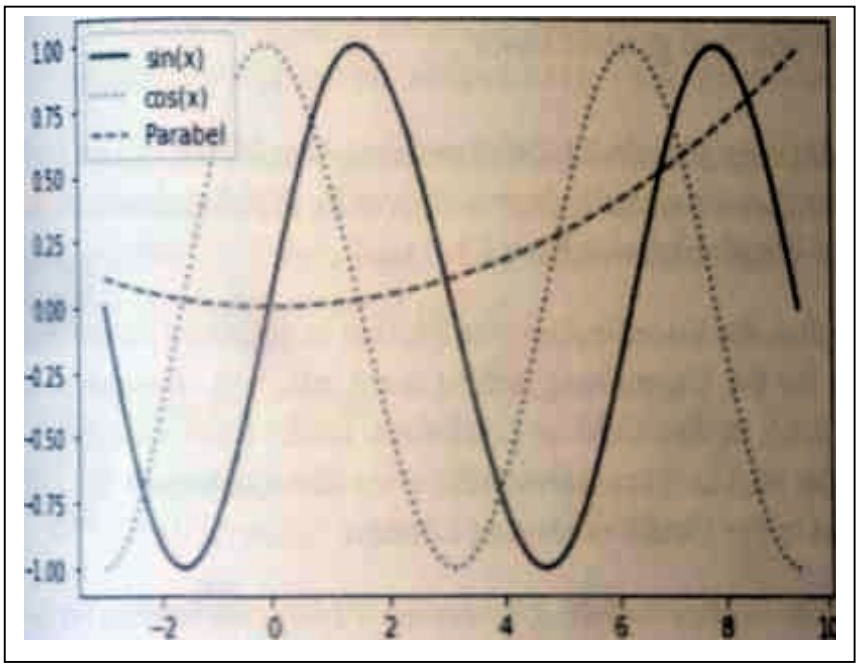


Figure 31.2: Curves with legend and title

With various options of the ***plt.legend*** command, you can also place the legend in other places in the image, just as you wish!

Next, I would like to show you how to create scatter graphics (point clouds) using “***scatter***”.

```
Import numpy es np
```

```
Import matplotl ib.pyplot es plt

rnd = np. random. RandomState( 42)
x = rnd.randn(70)
y = rnd.randn(70)
colors = rnd.randn(70)
sizes = rnd. randn (70) *1000
plt .scatter (x, y, c = colors, s = sizes, alpha = 0.4,
cmap = “plasma”)
plt.colorbar()

plt.show()
```

The random number generator is started in the first line ...

This is actually a pseudo-random number generator. This is because the sequence of numbers is only apparently random - similar to the decimal places of π. Using “Random State”, you can define a starting point so that your results can be reproduced by everyone when it matters. The random series is then always the same! However, it does not matter which specific number you use. However, according to the well-known novel “The Hitchhiker's Guide to the Galaxy” by Adams, many computer scientists choose the number 42. You know, the answer to the “question of life, the universe and all the rest” ...

The following 4 lines each produce 70 random numbers. The small “n” at the end of `randn` shows you that these are standard normally distributed random numbers. Mean value is always 0, variance (and standard deviation) 1.

The 70 elements of x and y should represent the coordinates of the points in our scatterplot. You can use “colors” to make the display nicely colorful. Admittedly, this is not very attractive in a presentation printed in black and white. I have also varied the sizes for you. Again, they are distributed as standard, but this time multiplied by 1000 so that you can see something in the graphic.

Then finally comes the actual scatter command. You already know the first parameters. There is also the ***alpha = 0.4***, which indicates the “transparency”. A value of 1.0 means “no transparency”, whereas 0.0 would be “total invisibility”. In between are interesting color transitions, for example with an alpha of 40 percent as in the example. There is also a parameter for the color map (***colormap, cmap***). In addition to “plasma”, you can also try out “inferno” or “magma”, and these are by no means all the possibilities. Have fun with it! On the right in Figure 31.3 you will even find a color scale (***colorbar***).

In addition to the 2-D graphics shown, you can also add another dimension using plot3D. You can also display histograms, pie charts, polar coordinate plots and 1,000 other things. In each case with only a minimal set of commands. As you can see, I can't stop raving about it.

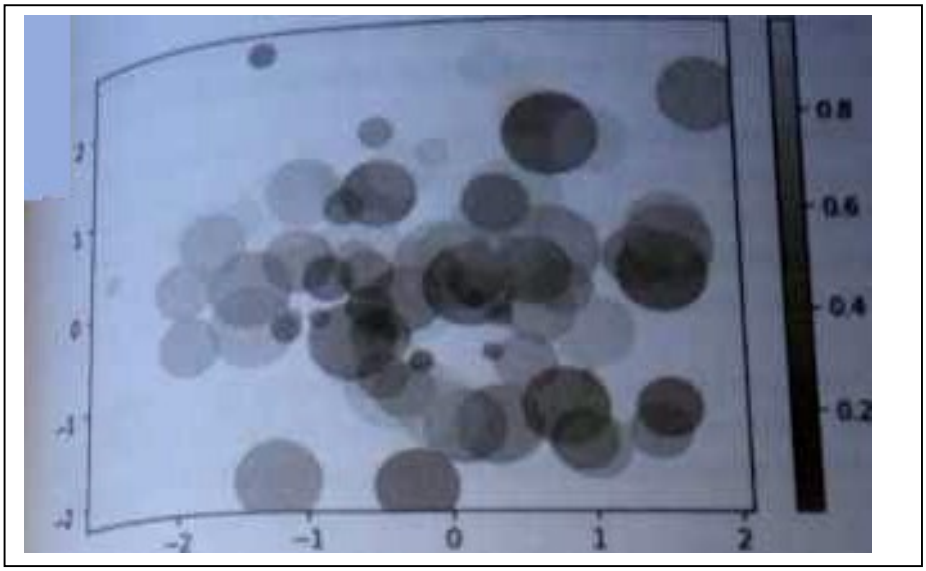


Figure 31.3: Example of a scatter plot

A nice overview of the various possibilities with many example graphics and the corresponding code can be found at

https: //matplotlib. org/stable/gallery/index

Serializing objects with Pickle

In programming practice, it can happen very quickly that you want to save objects that you may have created with a lot of effort (and load them again later).

In addition to the JSON format (JavaScript Object Notation) commonly used for general Internet exchange, for which a Python library is of course also available, I would like to introduce you to a simple and efficient Python-specific alternative: the package called ***pickle***!

```
Import pickle
dataobject = (“list”: [1,2,3,4], “number”: 17, “string”: “Python is great”,
“dict”: { 1: “one”, 2: “two”, 3: “three”))
file = open (“example .pickle”, “wb”)
pickle.dump (data object, file)
file.close()
```

In the example, the pickle package is imported first. With `dataobject`, I simply wanted to present you with a combination of nested data types, which can of course be much more complicated and much more extensive.

Use `open` to open a file called example. `pickle`. You are of course free to choose the name, path and suffix. The `wb` as the second argument stands for “write binary”. The file is thus opened for writing, and you should always specify the binary version for pickle!

The exciting line is the penultimate one: use `pickle. dump` to write the data object to the specified file, done! At the end, the file is closed again.

Of course you will want to read in your object again at some point:

```
file_new * open(“example. pickle”, “rb”)
data_object.new = pickle. load(file_new)
file_new.close()
```

I have now deliberately added “`_new`” to the names of the file and the object to be read in so that you can see that this could be done at a different point in a completely different program. This time the file is read in binary form “`read binary (rb)`” and with `pickle.load` into a Python object.

At the end, the read data object naturally looks exactly like the previously saved one:

```
{'List1: [1, 2, 3, 4],
'Number': 17,
'String': 'Python is great',
'Dict': (1: 'One*, 2: 'Two*, 3: 'Three')}
```

As Pickle - unlike JSON - is a non-readable form of any Python object, you should never read in third-party objects or objects from untrusted sources. Otherwise you could load all kinds of malicious code into your program!

Beyond the edge of the plate

It would not only be beyond the scope of this book, but an entire series, if I were to show you the full scope of all possible modules that you can integrate into your Python code.

So this last section of the chapter is just to give you a few more inspirations of what you can create in just a few lines with the right libraries. Or you can write your own modules and make them available to the public? This is generally very popular!

You can find a list of many modules that you have direct access to in Python at

```
https: //docs. python. org/3/py-modindex. html
```

SciPy

The scientific module (scientifiy python) contains a series of mathematical functions with which you can calculate derivatives and integrals, solve differential equations or connect points using ***BSplines***, for example. This is a polynomial train with which you interpolate piece by piece, but in such a way that you no longer recognize the transitions.

(because the connections have a common curvature and are therefore continuously differentiable).

In the example, the SciPy interpolation module is integrated first. It requires the functions splrep to find the BSpline representation of the points (with x and y coordinates) and splev to finally determine the smoothing polynomial including the derivative. The code is otherwise hopefully self-explanatory. The cosine is only used to create a few seemingly wildly scattered points.

```
Import scipy.interpolate as si
x = np.linspace(0, 10, 6)
y = np.cos(x)
spl = si.splrep(x, y)
x1 = np.1inspace(0, 10, 200)
y1 = s1.splev(x1, sp)l
plt. plot(x, y, “o”, color=“red”, label="single points")
plt.plot(x1, y1, color = “green”, label=“BSplines Interpolation”)
plt .xl im(-1 ,14)
plt.ylim(-1 .1,1.1)
plt.legend()
plt.show()
```

The result can be found in Figure 31.4.

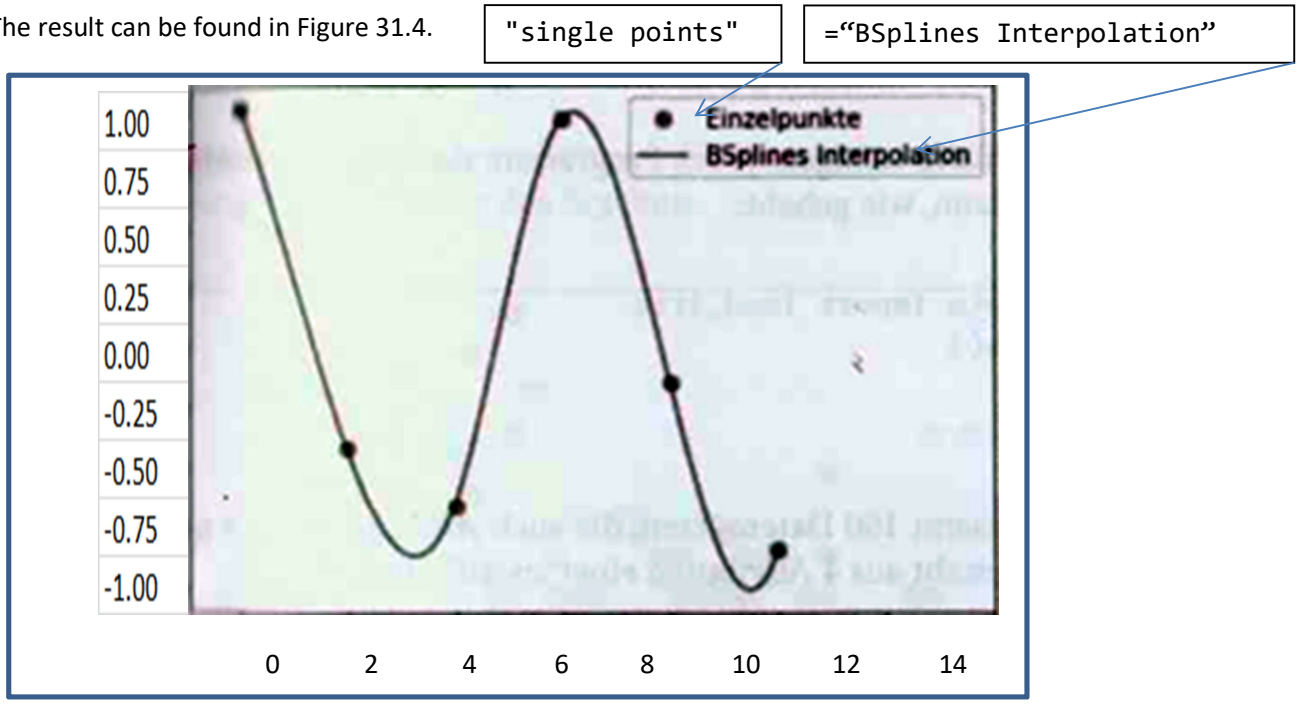


Figure 31.4: Interpolation of individual points using BSplines

scikit-learn

Artificial intelligence (AI) and machine learning in particular are on everyone's lips these days. Python plays an important role in this. In fact, it is thanks to the AI hype, among other things, that Python 3 has also achieved resounding success as the most widely used programming language.

Chapters 41 and 42 of the presentation deal with artificial intelligence, including machine learning.

Anyone new to machine learning and Python will not be able to avoid the scikit module. As the name suggests, you will find all possible algorithms for machine learning here, as well as a number of data sets to play with

A very comprehensive collection of algorithms for machine learning, with very good documentation, can be found at

```
https: //scikit-learn.org/stable/index.html
```

As an example, I will show you the “Iris” data set. The Greek word “iris” means “rainbow”. However, this is not about the iris in the eye, but about the flower genus of irises

The module is installed using

```
pip install scikit-learn
```

to install it. If you want to do this from a Jupyter notebook, please add an exclamation mark in front:

```
!pip install scikit-learn
```

This installation only needs to be done once. Every program that wants to use this library will then import it as usual:

```
from sklearn.datasets Import load_iris

iris.db = load_iris()

x = iris_db.data

y = iris_db.target
```

x is a NumPy array with a total of 150 data sets, which are also called attribute vectors. Each of these vectors consists of 4 attributes of a specific plant:

- sepal length in cm
- sepal width in cm
- Petal length in cm
- Width of the petal in cm

y is the set of assigned classes. You will find a total of 3 different species:

- Iris setosa, the bristly iris
- Iris-Versicolour, the variegated iris, which can also be found in medicines.

- Iris-Virginica the blue marsh iris

Actually, y is just a: NumPy array with also 150 elements, namely 50 times each the 0, the 1 and the 2. These correspond in this order to the different types of irises.

With

```
Print (iris_db.DESCR)
```

you can also have all these details written down.

It is a typical convention to denote the set of attribute vectors with a large X, while the set of associated classes is represented with a small y.

If you want to see how the goblet sizes relate to the associated classes, you only need a few lines of Python code:

```
Import matplotlib. pyplot as plt
plt.scatter(X:[: ,0] ,X[: , 1J , c = y, cmap="plasma",s=70, alpha=0.4)
plt.xlabel( "sepallength in cm" )
plt. ylabel( "sepal width in cm" )
plt. show( )
```

The code is another example of a scatter plot that you are already familiar with. Figure 31.5 shows the result.

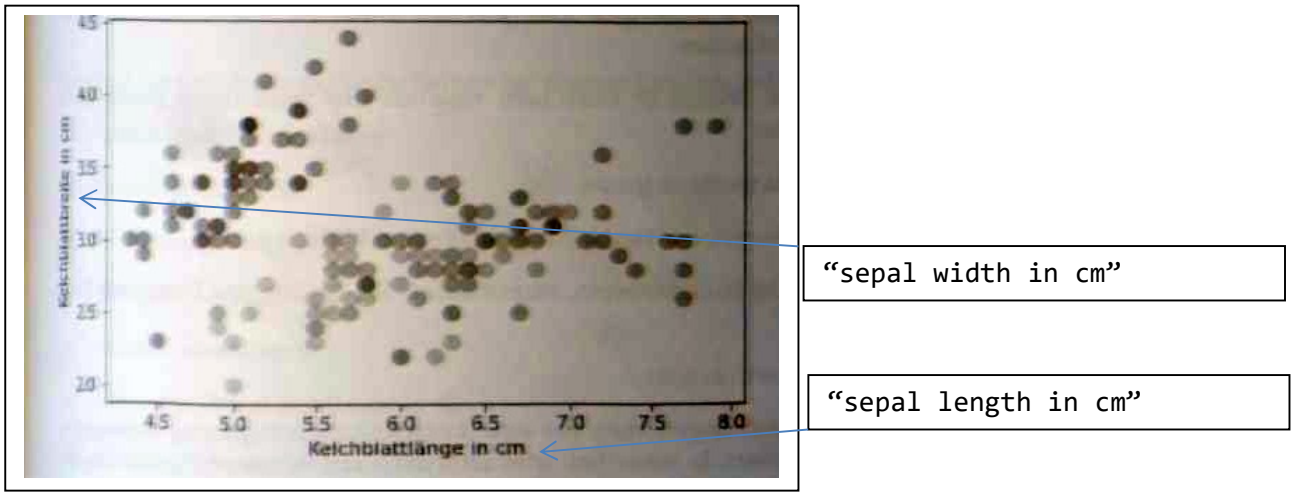


Figure 31.5: Distribution of sepal sizes of iris species

If you are interested in machine learning with this dataset, I recommend reading chapter 45!

Chess

It is said that chess has become particularly popular not only because of the pandemic situation, but also because of the Netflix series “The Queen's Gambit”. In any case, before that, with just a few lines of code, you could use all kinds of functions related to chess with the Python-Chess module.

Python-Chess also has extensive and highly recommended documentation:

```
https: //python-chess. readthedocs. io/en/latest/
```

After installing the module - preferably with `pip install chess` - you can get started immediately:

```
import numpy as np

import chess

board = chess. Board( ) # Starting position

move = np.random.choice(list(board . legal_moves))

board.push(move)

display(board)
```

In the third line, the chess board is initialized with the starting position. You then receive a random move from the list of possibilities created with the `legal_moves` generator.

The NumPy-function random, choice is very useful for this and will certainly serve you well in the future!

The move is then executed (`push`).

You can find a possible result of the output (`display`) in Figure 31.6.

If display does not work for you, please import explicitly in the Jupyter Notebook

```
from IPython.display import display
```

Otherwise, it may also be because you are using a development environment that requires other libraries. In any case, `print( board)` will work as an alternative and you will get a simple ASCII version of the chessboard.

With this method you can already write a simple chess engine that simply executes random moves - which admittedly does not lead to particularly interesting games...

Figure 31.6: Random chess position after White's first move

...and even more fun!

Surely you realize that we could go on and on like this: All possible areas are already covered by very many available Python modules.

A very extensive, though not exhaustive, collection of Python modules can be found at:

```
https: //pypi .org/
```

Just for fun, search for the following keywords under this link:

- Game development: Game
- Databases: SQL
- Web development: Web
- Wikipedia: wikipedia
- Chemistry: chemistry
- Everything you enjoy
- ...

If you find entering a keyword too laborious, you can also select the subject "***topic***" on the left-hand side. In most cases, the number of results is too large rather than too small; in this case, I recommend using a standard search engine and the corresponding hit to continue your research.

So, that's a little inspiration for now. If you want to know what makes Python tick, you can look forward to the next chapter...

Chapter 32 Becoming a snake charmer

In this chapter, we take a look at Python from the inside. At least we get very close to the core. To do this, we will extend Python with C code. This may not be open-heart surgery, but I promise you that you will gain a much better and deeper understanding of this wonderful language afterwards. If you have always wondered - coming from C - how you could be whisked away to the seventh heaven of the programming world so quickly and elegantly, you will find the answer in this chapter.

Time measurements

To get a more in-depth look at Python, it's important to delve into the deeper levels of execution. So far, every code has been more or less equally good for us.

For example, the NumPy library gives you the ability to apply functions to an entire list instead of a single element. Python calls this “universal functions”, or “ufunctions” for short.

If this topic sounds familiar to you, great! This was already mentioned in the previous chapter 31 in the section on ***vectorization***!

However, the real advantage of this ***vectorization*** is not that you save a few lines of code. Rather, your program will be dramatically faster!

To show you this, I will use the ***Timeit*** module in this chapter. You can add it to your source code using `import timeit` as usual. If you are using an]upyter notebook, the “magic command” `%timeit` is already built in.

Useful information on this and the other magic commands in the interactive IPython of the Jupyter Notebook can be found with the command

```
%magic
```

Let us measure the time required. In Figure 32.1, numpy is also imported and then an array a is created, which consists of the numbers from 0 to 999.

```
Import numpy as np
a = np.arange(1000)

%%timeit
b =[]
for element: in a:
b. append(element+1)

665 µs ± 809 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
%%timeit
b = a+1

2.82 ps ± 16.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Figure 32.1: Time advantage through vectorization

In the second section, the time is measured to create a list b that increments each element of a by 1.

The output means the following: The code was executed 1,000 times, taking an average of 665 microseconds. The standard deviation is 809 nanoseconds. That sounds pretty fast!

But if you take a look at the lower third, you can see how you can achieve the same goal with the vectorized version of "+": However, this time - on average - only 2.82 microseconds are needed. Because this is much faster, the process was repeated 100,000 times.

Of course, you can also determine the number of repetitions yourself. `timeit` calculates what makes the most sense.

Incidentally, it would of course not be a good idea to simply stand in front of the computer with your own stopwatch: Not only because the result would otherwise be heavily distorted by other running programs, even Python itself could distort the final result due to a very unfavorable garbage collection for data objects that are no longer referenced. If you use `timeit`, on the other hand, the garbage collection is automatically switched off during the measurement.

Okay, the speed advantage is quite nice. But where does it actually come from? Isn't the "+1" simply added to each element of a, as before?

To answer this very important question, which also allows us to take a closer look at the inner workings of Python, we want to add our own C function to Python. This can be done with the C extension,

C extension of Python

The reference implementation of Python is also called CPython. All the great things that C programmers only dream of are already included in Python, but they were ultimately programmed in C.

If you are wondering what the "C" is supposed to mean: The entire part V of your documentation is dedicated to this language!

So it makes sense to use the interface for C code, the C extension, to add your own C code to the Python functionality. This will also help you to understand Python much better.

CPython as a reference implementation of Python should not be confused with ***Cython***. The latter is a superset of Python, extended by C functionality. The extension is much easier to obtain there than here, but our aim is not to permanently combine Python and C in the source code. Instead, our considerations in this chapter are exclusively for a better understanding of Python.

To make an example that is a little more interesting, but at the same time not too complicated, our C extension should sum - similar to the example from the last section. However, the individual values are to be inverted beforehand. An ***inverse*** sum is calculated.

Python makes this easy to do:

```
def inverse_sum( 1 iste):
   sum = 0.0
   for element in list:
          sum += 1.0/element
   return sum
```

To translate this idea into C, the Python list must be translated into a C data structure. For the moment, we simplify this a little and pretend that we can pass the list directly as an untyped pointer.

`PyList_Size` is used to determine the length of the list. Otherwise, as is usual in C, we only have to declare all identifiers first and initialize them if necessary.

C-Extension first raw version, not yet compilable:

```
double inverse_sum(*list) {
       long len = PyList_Sise(1 is);
       long number;
       double sum = 0.0;

       for (long 1=0; i<len; i++) {
              zah1 = PyLong_AsLong(list[i]);
              sum += 1.0/number;
       }
       return sum;
}
```

In the main part - those were still the good old days of the classic for loop - we access the list as if it were a simple array, we also need to talk about this a little more. The number itself is converted to a long data type in the C code using `PyLong_AsLong`. The rest should be easy to understand. The essential core is the summation of the reciprocals - and that looks exactly like in Python. Oh yes: The annoying semicolons and many curly brackets are now of course on board! Instead, we have become so accustomed to the indentations - which are mandatory in Python but have no meaning in C - that we use them anyway.

However, for this code to play properly in Python, the extension macros and functions must first be inserted in C using =`include <Python. h>`. Incidentally, our new module to be imported into Python is to be called “dummies”. Therefore, the function must be renamed to dummies_invers_sum. Also, you are now ready for some more specifics of embedding in Python. It will look like this:

C extension, compilable version, file name: invers.summe.c

```
    include<Python.h>
    PyObject* dummies_invers_summe(PyObject* self, PyObject *args) {
       PyObject *list;
       if (!PyArg_ParseTuple(args, “0”, &list)) return NULL;
       Py.ssize_t len = PyList_Size(list);
       PyObject *actual_element;
       long number;
       double sum = 0.0;
       for (long i=0; i<len; i++) {
              current_item = PyList_GetItem(list, i);
              number = PyLong_AsLong(current_item);
              sum += 1.0/number;
       }
    return Py_BuildValue(“d”, sum);;
    }
```

In reality, the arguments are determined using a standard procedure, which you can see in the first lines.

In essence, the `PyArg_ParseTuple` does all the work for us. The second parameter O stands for object. Only then is the variable list actually provided with the transferred list

Of course, we cannot access this list as we can in Python. What were you thinking? We are in C. You have to do everything yourself. Fortunately, `PyList__GetItem` also helps. This is a bit more complicated syntactically, but not really difficult. The required index i is passed as the second parameter. So that it doesn't get so confusing, I have added the variable `current_item` of the type “pointer to PyObject”. I really hope that this makes the code easier to understand. Yes, even the good old pointers are now represented again.

After all, we are not returning a simple double data type either. Don't forget that this function is to be called later in your Python environment. Therefore, type-casting using `Py_BuildValue` is necessary. The “d” stands for “double” in C, but in the Python environment this will become a float.

Of course, that can't be everything. Somewhere you have to tell Python which function has been added. Also, you may have missed it already, where is our *docstring*?

The third and final version of the example contains the remaining necessary code. This should be added in the same file - by convention please call it *invers_summe.c* - directly after the compilable version:

C extension, remaining lines of code, file name: *invers_summe.c*

```
static PyMethodDef methods[] = {
   {“invers_summe”, dummies_invers_summe, METH_VARARGS, “Sum of the inverse"},
   {NULL, NULL, 0, NULL}
};
static struct PyModuleDef module = {
   PyModuleDef_HEAD_INIT,
   “dummies”,
   “Examples C_API”
   -i,
   Methods
};
PyMODINIT.FUNC
Pylnit_dummies(void) {
   return PyModule_Create(&module);
}
```

Essentially, you will find a list of all new functions of the module including docstring in methods []. The line with the many NULLs at the end marks the end of the list.

Welcome to the funny world of C!

The other two definitions register the additional functions in the module. You can then compile everything. The best way to do this is with a setup file (with the name setup.py):

```
from distutils.core Import setup, Extension
setup(name='dummies',
   version=s' 1.0' ,
   description='Dummies module',
   ext_modu1es=[
        Extension(“dummies” , [“invers_summe.c”])
   ]
   )
```

That was it already. Using

```
python stetup.py install
```

your Python module is now created. The name of the dynamic library created depends on the operating system. This should work on all common platforms. The prerequisite is that you have fully installed Python and have a functioning C compiler!

To try out the result, simply start the Python interpreter from the folder in which you have just created the module. Or navigate exactly there in the Jupyter Notebook. Then the fun can begin ...

The fun begins

From this moment on, you can use

Import dummies to use the new functionality in your Python environment. Let's try this out:

```
dummies.inverse_sum([1,2,3])
```

The result looks correct:

```
1.8333333333333333
```

Let's try this out with a slightly larger list and then compare the time required with the pure Python solution (Figure 32.2).

The effort was worth it: the C extension version of the summation is a thousand times faster than the pure Python code. Note the units: Microseconds versus milliseconds! And you can probably already guess why: The C code does it.

```
Import dummies
[i*i for i in range(1,100000)J

%timeit dummies. invers_.summe(l)
689 µs ± 6.71 µs per loop (mean ± std- dev. of 7 runs, 1,000 loops each)
def invers_sum (list) :
   sum = 0.0
   for element in list:
         sum +=1.0/element
   return sum
%timeit invers_sum(l)
5.63 ms ± 2.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Figure 32.2: Comparison of the inverse sums

It is nothing other than what you see there. The Python code, on the other hand, is much more complex, there are type checks and type conversions. In the C code, we simply assumed that we were dealing with integers. Python code, on the other hand, has to check this for every operation.

However, this is not the decisive difference. Even if you checked every element in the C variant using `PyLong_Check` and `Py_Err Occurred`, yes, even if you also made sure that there was no division by zero, the C code is still a hundred times faster.

The byte code

So far, we have not looked in detail at how Python works. The code is interpreted, isn't it? In fact, similar to Java, the Python code is translated into byte code. This in turn is executed on a virtual machine.

You probably want to know exactly what this looks like. To do this, we simply disassemble byte code using `dis` (Figure 32.3).

The result actually looks like assembler code.

Need a little refresher on assembler? Chapter 16 is waiting for you with the relevant information!

Of course, this cannot work for our C code:

TypeError: `don't know how to disassemble builtin_function_or_method objects`

You will also find the reference to the built-in function if you call up the help for our dummies function. Of course, you will not find any Python source code there either. In Figure 32.4 you will find the *docstring* from the C source file.

```
Import   dis
dis.dis(invers_sum)

  2        t LOAD CONST               1 (8.8)
           2 STORE EAST                1 (summe)

  3                                   8 (liste)
           4 LOAD FAST 6 SET ITER
         » 8 FOR~ITER                 18 (to 28)
          18 STORE FAST                2 (element)

  4       12 LOAD FAST                 1 (summe)
          14 LOAD CONST                2 (1.8)
          18 LOAO FAST                 2 (element)
          18 BINARY TRUE DIVIDE
          28 INPLACE ADD 22 STORE
          FAST                         1 (summe)
          24 JUMP ABSOLUTE             8

  6     » 28 LOAD FAST                 1 (summe)
          28 RETURNVALUE
```

Figure 32.3: Disassembly

```
help(dummies.invers_sum}

Help on built-in function invers_sum in module dummies:

Inver_sum(...)

   Sum of the Inverse
```

Figure 32.4: Help on built-in functions

Speed of light with vectorization

This insight was already quite useful. Python code is nowhere near as efficient as pure C code because the byte code is generated first, which in turn runs on the Python virtual machine. The routines of the C extension are excluded from this because there is no byte code for them and they run natively, so to speak. So far, so good. But what benefits do you get from vectorization with the universal functions?

Let's put this to the test. To be able to use vectorization, the list of numbers is first converted into a NumPy array. This is followed by vectorized inversion and finally the sum. There are two options for this

- the sum () function built into Python
- the NumPy function np.sum()

You can find the results in Figure 32.5. For orientation purposes, I have also run our C extension again without vectorization (box 3). The deviation from the last run is within the usual range.

The result of the NumPy vectorization (box 5 with the built-in function sum () and box 6 with the corresponding Numpy function) is very fast, but does not come close to our native code. There is a difference of 3 orders of magnitude. However, the question arises as to what proportion of the total consumption the additional conversion of the list into a NumPy array had.

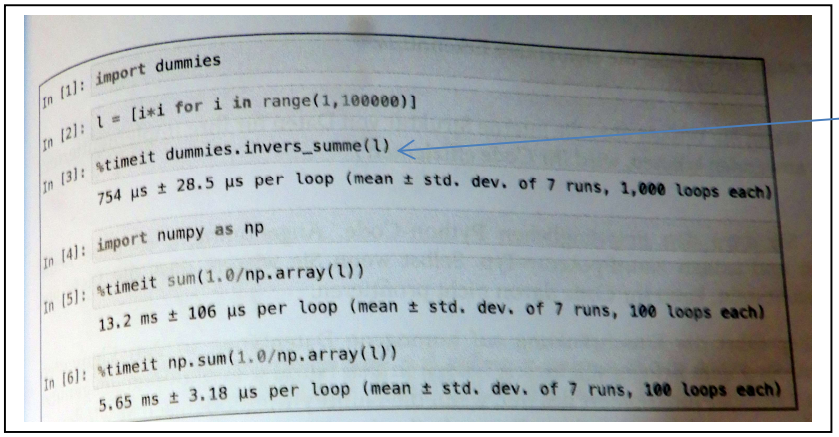


Figure 32.5: Comparison with vectorized sums

Figure 32.6 shows the result when starting with the correct data type:

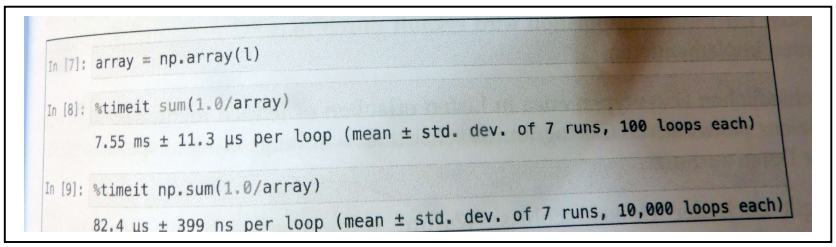


Figure 32.6: Vectorization to NumPy array

In the seventh box, array is immediately created as a NumPy array. Further summation takes place on this data structure. There is a clear winner: If the conversion is omitted, the NumPy sum is faster than the native C code by a factor of 9!

What seems surprising at first glance becomes obvious as soon as you realize what the difference is between an ordinary Python list and a NumPy array. A list consists of arbitrary objects. You can imagine that - in terms of the CPython implementation - it consists of a set of pointers to Python objects.

In contrast, the NumPy array consists only of homogeneous data types. These can be implemented as a C array of pure numbers, as in the example.

A type check no longer needs to take place. The space required per entry is always the same. The vectorized inversion is very fast.

Interestingly, the application of the usual Python sum (which our old problem with the different types expects anyway) is even slightly slower than the dummy inverse sum. This should be resolved if the checks mentioned above are performed in the C code.

Here, too, the elementary realization becomes apparent:

If you can apply knowledge about the internal structure of data to your implementation, your code will be more efficient.

Consider the original Python code. Suppose there was no C extension and no NumPy array type. Even if you know that the list consists of nothing but numbers, your code cannot benefit from this.

On the other hand, the restriction to homogeneous data types in the elements only leads to success because this knowledge can be utilized in the NumPy functions. Obviously, this only works if the NumPy methods themselves are not written in Python, but are part of CPython or the C extension.

Equipped with this knowledge, you now have a better understanding of where Python fits into the colorful world of programming languages: The great convenience with the various possibilities for data manipulation comes at the price of hard C programs that implement these possibilities.

However, the different data types in lists, for example, do not allow the code to keep pace with the efficiency of a native C program - as long as it does not exploit special properties of the data.

However, the reverse is also true: Why bother with pointers and hardware-oriented programming if the current requirements do not need (maximally) efficient code in the first place? So the golden way is to use Python wherever the elegance of the program is important and to pimp it (with C extension) where speed is really important.

This is the reason why you also come across Python programs in areas where maximum efficiency is important, such as machine learning. It's not immediately obvious from the libraries that the methods that are crucial for performance were programmed in C. But you know that now...

But you know that now...!

Got an appetite for machine learning? Why not continue with chapter 43 right away!

What this chapter is about:

To shed light on the concept of “artificial intelligence” (AI)

Recap important achievements of the AI

Answering small and big questions of the AI

Understanding the fundamental problem of AI

Chapter 41 Guided tour through the evidence chamber

This introductory chapter to the fascinating field of artificial intelligence aims to give you an impression of how colorful and diverse this branch of knowledge is. Many science fiction authors throughout the ages have explored the possibilities of artificial intelligence. As you read, you will understand what is actually feasible and what will - forever - remain fiction. Everything in between is incredibly gripping, see for yourself!

On the trail of cyborgs

Most people associate artificial intelligence or AI for short, with more or less frightening images of computers that will sooner or later take over the world. They have the feeling that, over time, the performance of computers will eventually exceed that of our brains. Cyborgs will eventually replace humans completely.

Others, however, believe that the entire scientific field of AI is garbage. Although researchers are working on transferring the cognitive abilities of humans to machines, “they are fundamentally failing. According to this world view, this is because our brain does not function in the same way as a computer and can therefore never be replaced by a machine.

As you can imagine, the truth lies somewhere in between. In the next few sections, I will show you that AI can actually achieve breathtaking and almost revolutionary progress over the last few decades. However, it has always failed in its most fundamental objective. This is also known as strong AI.

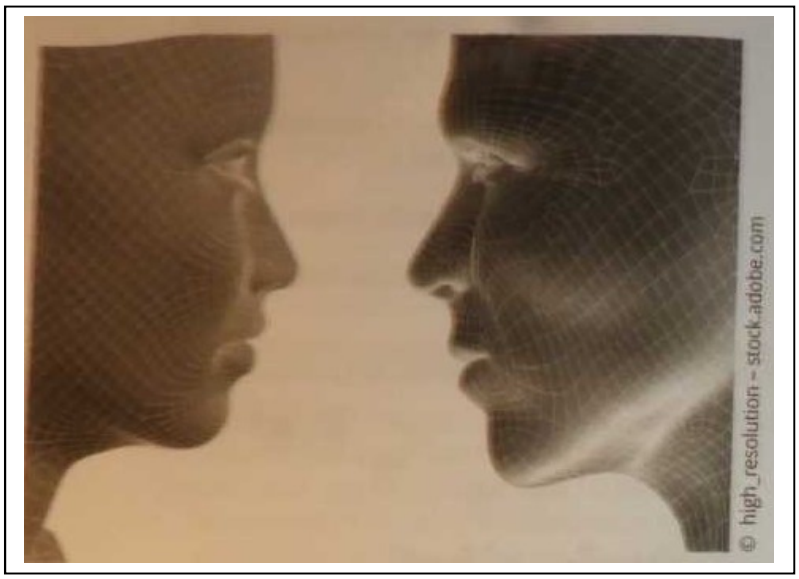


Figure 41.1: Human-like machines: utopia, nightmare or nonsense?

Strong AI attempts to emulate human intelligence. This includes all cognitive abilities such as creativity, learning, problem-solving, remembering, imagining, planning, reasoning, perceiving, orienting, wanting and believing, right up to having its own consciousness. Emotions are also to be transferred to machines as part of strong AI.

The problem with strong AI is that it wants too much. As early as 1950, Alan Turing proposed a test that could be used to prove strong AI.

You can find out more about Alan Turing in chapter 54.

This Turing test is quite simple. You are sitting at a computer and chatting with a person. If you are absolutely sure that the person is a human being, even though in reality there is only a computer program on the other side, you have passed the Turing test, otherwise not. To date, no program has passed this test - carried out by a recognized panel of experts. However, language models such as ChatGPT, which have been freely available since the end of November 2022, come surprisingly close to achieving this goal!

However, it would be unfair to reduce AI to strong AI. Let's shift down a gear and talk about weak AI.

Weak AI is content with solving specific, difficult application problems. Examples would be playing chess, perceiving objects in rooms, finding solutions in labyrinths, autonomous driving, swimming, flying, recognizing faces, proving mathematical theorems, planning tasks, building towers. The list is endless. Anything that - at least in appearance - requires a certain amount of intelligence is suitable as a task.

The weak AI is extremely successful in almost all areas it has set itself. While the positive answer to the strong AI question also entails extremely complex philosophical questions, the weak AI is relieved of this burden.

If this continues - and we can assume it will - it will have two very interesting consequences:

- Computers will never take over the world.
- You don't need to be afraid of cyborgs replacing you at some point.

However, I admit that there are already certain areas of humanity that are controlled by machines to a frightening extent. In the stock market, computers “sit” at the levers for super-fast decisions, which have already triggered one or two crashes because they fueled each other's downward trend. However, this could also fall into the category of “immature software”. Aircraft have collided with others despite collision warning systems. On the other hand, autopilots usually work more reliably than human ones, as long as no unforeseen (by the programmers) events occur.

At the end of the last century, Barry Kasparov, the undisputed world chess champion, set out to defend humanity's (intelligence) against AI. The result is not surprising for experts. Although he put up a good fight, world chess supremacy now clearly belongs to the computers. But that is not frightening. In contrast to strong AI, which tries to create chess programs based on the model of human grandmasters, today's chess computers work as the representatives of weak AI imagined: extremely reliably, but completely differently to the way humans do. Grandmasters select very few moves from a vast number of possible moves and intuitively assess the resulting positions. The fastest chess computers, on the other hand, calculate more than a billion positions per second and mindlessly count the material. A billion positions - that is an order of magnitude far beyond what a human player can evaluate in a lifetime.

This example clearly shows the difference between strong and weak AI. Advocates of strong AI ridicule today's chess programs and sometimes deny that they are part of AI at all. Rather, it is a purely technical-mathematical solution to a problem, which has nothing to do with intelligence.

At its core, this also represents a major dilemma for AI: The moment a solution algorithm has been found for a specific problem, many deny that intelligence is required to solve it. This means that the problem does not even fall within the remit of AI.

I don't want to bother you too much with this rather complicated discussion. However, it is important in order to define and delimit the field of AI appropriately. Nowadays, some people even associate “machine learning”, the origin of which is clearly an AI question, with the field of engineering. But don't let that stop you: If a problem interests you, just apply AI methods to solve it once...!

The term Artificial Intelligence (AI) goes back to a conference that was convened in 1956 by John McCarthy and some of his colleagues as part of a research project. You should not forget that the term intelligence is much broader in English than in German. Just think of the “Central Intelligence Agency” (CIA), which is not the central intelligence agency of the USA, but is much more generally entrusted with information procurement and processing (foreign intelligence).

Knowledge without conscience

One of the great insights of AI is that learning is generally hardly possible without already possessing an enormous amount of knowledge. The original idea was to simply build a system that was able to increase its own knowledge. If the program were only complicated enough, it would become intelligent by itself, so to speak. Unfortunately, this is an illusion.

Efficient knowledge acquisition requires a considerable amount of knowledge in advance Even newborn babies have an enormous variety of skills and abilities that are indispensable for the further acquisition of knowledge. Of course, the child has already had nine months in its mother's womb to acquire this knowledge - without having to do anything!

Knowledge-based systems are an important pillar of artificial intelligence. One subclass is particularly important: expert systems that process the knowledge of human experts.

Chapter 44 is all about expert systems.

Planning and decision-making

Planning and decision-making are among the cognitive abilities that we like to associate with intelligence - at least if the plans are well thought out and the decisions turn out to be correct.

Logical reasoning is understood as an underlying property that is a prerequisite for meaningful planning and decisions. Fortunately, logic has already been analyzed mathematically, completely independently of the AI. There is even a programming language that is based on logical reasoning: ***Prolog.***

Chapter 53 contains an outline of the branch of knowledge known as logic. An introduction to Prolog is provided at the beginning of Chapter 44.

Pattern analysis and recognition

When we talk about facial recognition in this country, we usually think of surveillance and threats. From a scientific point of view, pattern recognition is a prime example of successful AI.

You should define the term “pattern” very broadly. Any collection of objects with certain properties can contain a pattern that can be learned.

However, patterns can only be recognized once they have been learned. Learning systems are perhaps the most fascinating area of AI.

The entire chapter 43 deals with learning systems.

Intelligent agents or search or what?

Where are “Google” and the other search engine manufacturers? Isn't that AI too? Knowledge is acquired on the web via agents, small programs that are able to solve tasks independently. The question is even more fundamental: how can problems be solved using AI in general?

A first (and early) answer was provided by the concept of the General Problem Solver (GPS). The idea was to develop an algorithm that could solve any problem. This claim was also too ambitious. Nevertheless, it indirectly produced excellent results for methods for intelligently searching a problem space. In general, the issue of searching is very closely linked to artificial intelligence. Isn't problem solving basically a kind of “search”? Doesn't the intelligence of the search consist precisely in the fact that a - possibly almost unmanageable - large selection of options for action (in the search space) is searched through in a clever way? You can indeed see it that way.

The basics of GPS and some important algorithms for the search can be found in the following chapter 42.

Artificial beings with their own consciousness

Well, in this last section I would like to come back to strong AI.

To be honest, nobody knows what consciousness is these days, at least from a technical point of view.

It is therefore pointless to think about how consciousness can be artificially created using computer programs.

Similar to the Turing test, you can never know whether a machine has actually developed its own consciousness. If you ask a program a question and get an answer that looks as if it is conscious, you can never know whether it really is.

In the same way, you can never know from your fellow human beings. However, we tacitly assume that we are all at least similar enough to each other to feel the same way about this question. I believe myself to have a consciousness and therefore I assume that this is also the case for everyone else.

But why is that ultimately the case? Perhaps the “computing power” of our brain, which is far superior to that of ordinary computers? Perhaps...

Back in the last century, the rather obvious idea was pursued of simply recreating our human brain using computers in order to develop a better understanding of ourselves. Once the functioning of a single nerve cell in the brain was understood, these artificial neurons simply had to be connected to form a network. The result was research into artificial neural networks.

Neural networks are the exclusive topic of Chapter 45.

At the time, however, this failed due to the sheer volume: reproducing the more than 80 billion nerve cells of a human brain with computers is no small feat.

That was back then. Today there is the European Union's Human Brain Project. The idea is that if we all join forces, we should be able to generate the necessary computing power together. That should be the case. Ambitious goals want to model one hundred trillion neuronal connections in a human brain.

Was passiert aber, wenn dieses Programm zu einem »Wesen«, gar mit Bewusstsein werden sollte?

Meine Meinung dazu kennen Sie ja bereits. Dennoch ist das Projekt faszinierend und inzwischen sogar ... abgeschlossen!.

Alle Details zum Human Brain Project finden Sie unter:

`https://www.humanbrainproject. eu`

42 Searching and finding through play

The word “artificial intelligence” has an almost esoteric ring to it. However, after reading this chapter, you will realize that there are hard engineering methods behind it. Here you will find a selection of important algorithms for systematically solving problems. You will also come frighteningly close to the world formula for solving any problem ...

Tracking with GPS

Forget everything you've ever heard about computer science. What is the use of computers all about?

In very abstract terms, computers are supposed to help you solve problems.

How can this - again in principle - even happen?

Someone - in case of doubt you (!) - has to tell the computer what the initial situation is, what possible courses of action are available and what you would accept as a solution.

Imagine a game of chess (or any other board game).

- The starting position is the initial position of the pieces.
- Possible courses of action are determined by the rules of the game: Which piece can move or capture other pieces and how?

- A solution in this case would be a positive outcome, such as a mate position for the opponent.

However, not all problems can be modeled as games. The problem often consists of a ***classification*** task

Suppose a program is to identify a face on a digital photo.

- The starting point would be a set of photos that show different faces and whose classification (i.e. the identification of people in the photos) is already known.
- Options for action would be methods for extracting various relevant features from the bits. From this, the features should be assigned to the correct (already known) faces.
- The solution would of course be to assign previously unknown digital photos to the correct persons.

Although you can interpret many problems as classification tasks, this does not apply to all conceivable situations. Think of ***robots***, for example.

A ***mobile robot*** has to find an exit from a labyrinth.

- The starting position would be the exact starting position of the robot within the given maze.
- Options for action would arise from the branching possibilities at each intersection.
- Of course, the solution would be found as soon as the robot passes the exit.

As you can see, these three examples can be solved in the same way, even though they appear to be very different. To do this, start with a problem space.

You can imagine a problem space as a directed state graph whose initial node corresponds to the initial position. Each edge leads to a new state. The number of edges results from the respective number of action options per state. The solution is one or more terminal nodes (desired end states).

Think about this for a moment: Actually, the problem has lost its scare factor once you realize that it can basically be understood as finding a path from the initial node to a terminal node.

You can find out all about graphs in Chapter 37.

Imagine that this state graph is a problem space. The initial state is the initial situation. The desired solution is the terminal state. The possible courses of action are marked by arrows. The human observer finds the solution quite quickly via the states. But if you take a “wrong turn” in state Z8, you have a problem: you can then carry out further action steps in the solution space for an infinitely long time without ever reaching a desired target state.

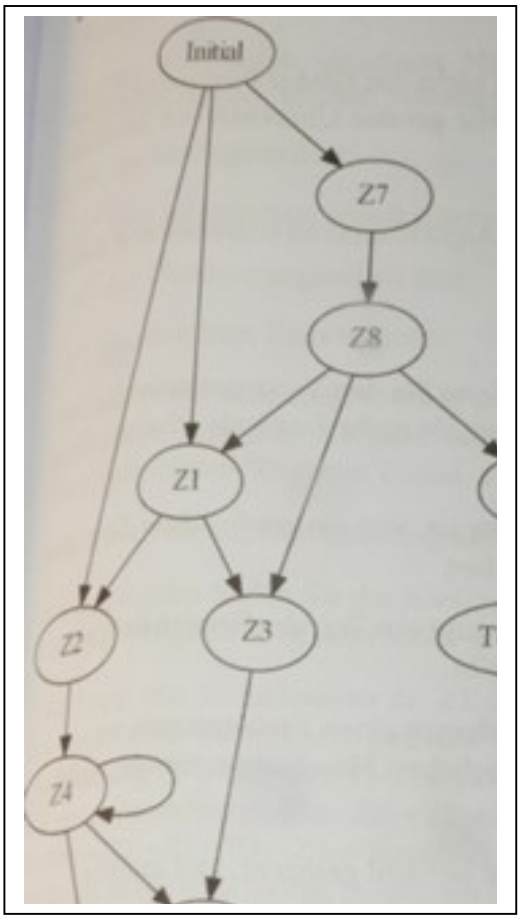


Figure 42.1: Exemplary problem space

You don't believe me that something like this can happen in reality? Take the Rubik's cube as an example. Its initial state corresponds to a twisted position. You could rotate it for as long as you like without ever achieving order in the side coloring if you do not proceed systematically.

However, if you succeed in systematically finding a path from the initial node to the terminal node for every problem space, you will be able to solve any problem that can be modeled in this way.

A General Problem Solver (GPS), with which you can - theoretically - systematically solve a very large number of different problems.

An example of a GPS would therefore be an algorithm that systematically searches through a problem space and finds a path from the start node to an end node, assuming that such a path even exists.

Of course, GPS in this chapter does not refer to the Global Positioning System, in which navigation devices use very precise times to determine their positions on the earth very accurately.

Who is stopping you from writing your own GPS algorithm today and solving humanity's problems once and for all?

Well, there are a few hurdles ...

- Some problem spaces are insanely large. If you want to search through the entire chess variant tree, you will end up with more states than the known universe has atoms.
- There are problems in which it is not at all clear what a desired terminal state is. Politicians argue about this every day.
- For a number of problems it is not known whether a terminal node exists at all, and if it does, whether there is actually a way to get there.
- Sometimes the difficulty lies in constructing a problem space at all. For some states, the possible courses of action are quite confusing.
- If it were that easy, someone would have done it long ago and there would no longer be any human problems at all ...

Nevertheless, an incredible number of problems can be solved in this way.

The mountaineering method

Sometimes it's even quite simple. Imagine you want to climb a mountain. This is the problem. You are at the foot of the mountain, which is the starting point. The solution would be reached when you are at the top. Naturally, you want to get closer to the top with every step you take.

The mountaineering method for solving problems is to always choose the option that lets you climb the highest

That may sound good, but with real mountains it can happen that you first reach a summit and then have to take a downhill route before you finally reach the top. This also applies to certain problem classes. The mountaineering method must therefore be refined. You have to accept that you will have to get worse in the meantime in order to reach the highest goal at the end.

However, there is usually another difficulty in addition to general problem solving: costs, in the general sense of the word. The costs of each individual action option of a problem (or the arrow of a state graph) can consist of ...

- the time required,
- the technical/physical effort (including energy requirements),
- the persuasion work and
- a real price in euros.

In this case, you write a number representing the costs on each edge of the node, and your task is to find not just any but the most cost-effective path from the initial to the terminal state.

Take a look at Figure 42.2.

At the edges you will find the costs for the respective state transition. Can you find the cheapest path from the initial node to the terminal node?

The price from initial node to Z1 is cheaper via Z2 because 3 + 3 is less than 7. It is even worse because the path via Z2->Z1 to Z3 is cheaper than the direct connection (3 + 3 + 2 < 9). Using the same argument, you find that the cheapest route to Z4 is via Z2->Z1->Z3. This also results in the optimal solution: Initial->Z2->Z1->Z3->Z4->Terminal with a total cost of 3 + 3 + 2 + 2 + 3 = 13. All other paths are more expensive, try it out!

Did you see that there is also a path from Z2 to itself? These are meaningless costs, but they do occur in practice (for example, this corresponds to fuel costs when the engine is running).

To systematically explore such paths, it is best to use a ***greedy*** algorithm. You simply always choose the path to any node that causes the lowest costs.

A ***greedy algorithm*** attempts to minimize the costs up to the destination in each individual step.

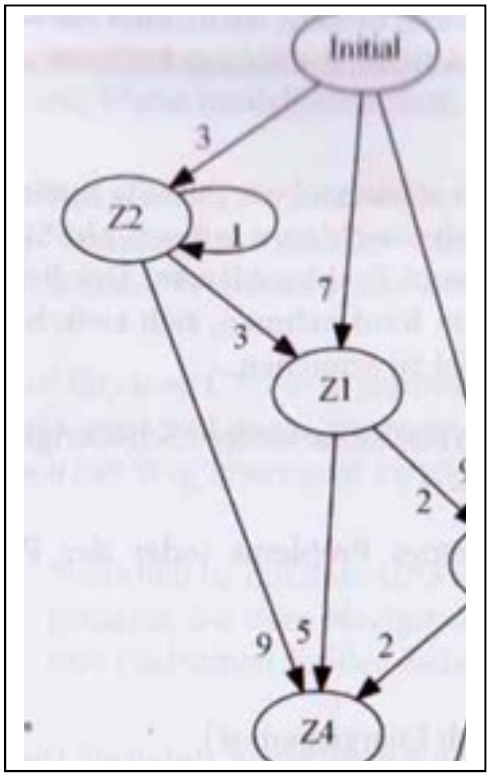


Figure 42.2: State graph with cost labeling

As an example, I will describe the ***Dijkstra algorithm***:

This algorithm is also mentioned in chapter 37 of your book

1. you create a set of open (i.e. still unhandled) nodes OK. At the beginning, this set consists only of the initial node. You also create an initially empty set of treated nodes BK. A node value is assigned to each node. Only the start node is initialized with a zero, all others are initialized with an artificial maximum value ∞ (infinite).

2. select the node with the minimum cost value from the OK set. If this is the terminal node, you are already done.

3. otherwise, calculate the costs of all subsequent nodes for this node that do not already occur in BK and enter them in OK. The costs of a node are only updated if the new value is smaller than the existing value.

4. the originally analyzed node from OK is moved to BK.

5. continue with step 2.

And that's it already. As an example, I will show you how this greedy algorithm works in the case of the graph in Figure 42.2.

The sets OK and BK are initialized as follows: OK = { (Initial,0)},BK = { }

In the first step, the nodes Z1, Z2 and Z3 are regarded as immediate successor nodes of the start node and packed into OK. None of these nodes occur in BK. The values are simply derived from the edge weights. The initial node is moved to BK:

OK = { (Z1,7),(Z2,3),(Z3,9) },BK = { (Initial,0) }

The smallest value of a node in OK can be found at Z2. Possible subsequent nodes are Z2, Z1 and Z4. The value of Z1 is updated because 3 (the cost of Z2) plus 3 (the edge weight between Z2 and Z1) is less than 7. Z4 is also updated with 3 + 9 < ∞. In contrast, the following applies to Z2:3 + 1 > 3, so no adjustment is made here. Z2 is then moved to BK:

OK = { (Z1, 6), (Z3,9), (Z4,12) }, BK = { (Initial, 0), (Z2,3) }

We continue with Z1, currently the smallest element in OK. Its successors are Z3 and Z4. Both nodes are adjusted because 6 + 2 < 9 and 6 + 5 < 12. You get:

OK = {(Z3,8),(Z4,11) }, BK = { (Initial,0),(Z2,3),(Z1,6) }

Now it is Z3's turn. Successors are Z4 and Terminal. As 8 + 2 < 11, Z4 is adjusted. The value of Terminal results in 8 + 6 = 14.

OK = { (Z4,10),(Terminal, 14) }, BK = { (Initial,0),(Z2,3),(Z1,6),(Z3,8) }

As you can see, the end node has already appeared in OK, but before that, it is Z4's turn because 10 < 14. The subsequent node of Z4 is only Terminal. The costs result in 10 + 3 = 13, which is less than 14:

OK = { (terminal, 13) }, BK = { (initial, 0), (Z2,3), (Z1,6),(Z3,8), (Z4,10)}

Next up is Terminal. As this is the terminal node, the algorithm ends at this point. The costs from the initial node to the terminal node result from the already known sequence -->Z2->Z1->Z3->Z4-> and add up to 13.

Heuristic search in the hay

However, a general problem solver cannot assume that the complete solution space is already available as a graph. Instead, the algorithm must first build up the tree during its processing. To do this, the possible subsequent nodes are expanded when a node is reached (for the first time).

If it is a game, each node represents ijl position. The subsequent nodes result from the possible moves defined by the rules of the game. This part of the algorithm is also called the move generator.

The Dijkstra algorithm from the last section actually provides the optimal final result, provided that there are no negative edge weights. Consider Figure 42.3.

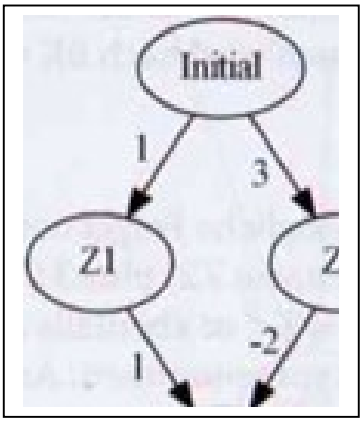


Figure 42.3: Graph with negative edge weights

The fast greedy algorithm finds the solution of the path Initial->Z1->Terminal with a cost of 2. Node Z2 is never expanded because its cost 3 is higher than that of the terminal node. However, the path Initial->Z2->Terminal with a value of 3-2=1 is actually cheaper.

The search costs for the “greedy” best search are always minimal. The search itself is not optimal (with possible negative edge weights).

To overcome the problem of negative costs, you should try something new, a heuristic!

A heuristic (from the Greek *heuriskein*, *to find*) is a method of finding the solution to a problem by clever guessing rather than exact calculation.

Heuristics are extremely important in practice. We use them constantly without always being aware of it.

- We try to find the quickest way to our destination without calculating it in detail every time using maps.
- We buy items because we think they are cheap, even though we can't be sure.

- We accept a job without knowing whether this choice is really optimal.

Archimedes of Syracuse is said to have exclaimed “Eureka” (“I have found (it)”) when he discovered a method for measuring the volume of any body in a bathtub. However, this is not a heuristic: the amount of water displaced corresponds mathematically exactly to the volume.

With a heuristic, you avoid time-consuming calculations and instead - hopefully - arrive at the solution immediately and intelligently.

A famous example of a heuristic is the airline distance for estimating the kilometers of road between two locations. Instead of measuring curved roads on the map, it is much quicker and more convenient to determine the distance between places. This usually works very well. Typical sources of error are mountains, rivers or missing roads on the way to the destination.

To solve the problem of the most cost-effective route, you now also apply a heuristic. This should estimate the costs to the destination.

The heuristic **h(n)** estimates the costs of node n for the cheapest route to the terminal node.

In contrast, **g(n)** calculates the actual costs required for the path from the initial node to node n.

The value of a node **f(n)** should result from the sum of the actual costs up to this point plus the expected costs up to the destination: **f(n) = g(n) + h(n)**.

This shows you the weakness of Dijkstra's algorithm from the last section:

The greedy method identifies the value of a node n with its actual cost: **f(n) = g(n).** In contrast, the estimated costs up to the target are neglected **(h(n) = 0).**

To find an optimal solution even with negative edge weights, you need an admissible heuristic that never overestimates the costs.

A heuristic is called admissible if it does not overestimate the costs up to the terminal node for any node in the graph.

On the other hand, h(n) may well underestimate the costs. For example, the airline heuristic is permissible: The airline distance between two locations is always less than (or equal to) the shortest road connection.

Navigating to the stars with the A* algorithm

To take the heuristic into account appropriately, you need a new algorithm called A* (pronounced "A-star"). In addition to the easily determined cost g(n) from the starting node to node n, a heuristic is now also used to calculate the value of the node f(n). However, save g(n) and h(n) separately, because only g(n) is needed for successors, while you determine the heuristic h(n) anew each time.

1 OK = {(Initial, 0)} BK = {}

2. determine the node n with minimum value f(n) from OK. Note that this value is now made up of the actual costs g(n) up to this point plus the heuristically estimated cost function h(n). f(n) = g(n) + h(n)

3. if n is the terminal node, the algorithm terminates.

4 Otherwise, n is moved from OK to BK.

5. perform the following steps for each successor node k of n:

- Update the value of k if necessary.

- If k is already in BK and its value has been adjusted in (i), all successors of k may also need to be updated. This also applies to their successors and so on...

- Insert k in OK if it is neither in OK nor in BK.

6. continue with step 2.

This ingenious extension is best demonstrated with the graph in Figure 42.3. As heuristic h(n) I use the constant function h(n) = - 4 for all nodes except the end node. I will explain how I arrive at this later. Logically, h(terminal) = 0 applies to the terminal node.

The initial state is as always: OK = {(Initial, 0)}, BK = {}

There are two successors of OK, namely Z1 and Z2. Due to the heuristic, their costs result in 1 + (- 4) = - 3 and 3 + (- 4) = ~ 1: I

OK = { (Z1, 1 + (-4)),(Z2, 3 + (-4)) }, BK = { (Initial,0)}

The smaller value is -3, so Z1 is next in line. There is only one successor, namely the terminal node. For the terminal node, each heuristic is always 0, which obviously corresponds to the cost until exactly this node is reached. You get:

OK = { (Z2,3 + (-4)), (Terminal, 2 + 0) }, BK = { (Initial, 0), (Z1, 1) }

This time, the value of Z2 is the smaller one compared to the terminal node: - 1 < 0. It is therefore Z2's turn. The value of the end node is updated because you have found a new, cheaper way via Z2:

OK = { (terminal, -1 + 0) }, BK = { (initial, 0), (ZI, 1), (Z2,3) }

Per terminal node is the only one remaining in OK, the algorithm terminates with the correct result: Initial->Z2-> Terminal.

Have you noticed why the A* has found an optimal solution? The heuristic of minus 4 did not overestimate the actual costs (minimum -2) at any point, but always underestimated them. Any other number less than minus 2 would of course have done as well. Compare the procedure with Dijkstra's algorithm: there, by neglecting the heuristic, h(n) = 0 is implicitly applied, which, however, overestimates the actual costs in the case of negative edge weights.

The A * algorithm finds the optimal path in a directed graph if it uses an admissible heuristic.

But the choice of heuristic is also of decisive importance for the behavior of the algorithm. I have listed how the A* algorithm behaves with different cost and heuristic functions.

- For h(n) = 0, this is a greedy algorithm.
- For h(n) = -∞, the optimal solution is always found, but the method converges very slowly.
- If h(n) always estimates the costs perfectly, the method directly delivers the optimal path without a search.
- The better h(n) estimates the actual costs, the faster the A* algorithm terminates.

Fun with MINIMAX and Moritz

The A* algorithm therefore leaves nothing to be desired. However, the search for a suitable heuristic is an extremely difficult matter that requires the maximum creativity of the computer scientist.

In principle, however, the method no longer works if you want to model two-person games. Take the game of chess, for example. The tree of variants is again a directed graph, but this time you are not simply looking for the shortest path to your opponent's mate. That would require your opponent to help you!

Because both opponents move alternately, the evaluation of the positions is reversed after each move. The cost g(n) that led to this knot, on the other hand, is irrelevant.

Assume an arbitrary chess position. Let's say you have the white pieces and it's your move. Of course, you choose a maneuver that is to your advantage - or to your opponent's disadvantage, which is the same thing. However, your opponent thinks the same way! Your task is to maximize the position evaluation, while your opponent assumes the opposite and tries to minimize your position.

If you assume that your opponent's play is optimal (but undesirable from your point of view), you should use a mini-max method when calculating variations. In doing so, you maximize your possible position evaluation, while your opponent minimizes your evaluation as soon as it is his turn. In any case, this is a good idea if you are up against a strong player.

The mini-max method always selects the move that maximizes your evaluation, assuming that your opponent will then minimize the evaluation of the new position. The decay is applied recursively to all subsequent positions.

Sounds complicated? It's best to look at a small example (Figure 42.4)

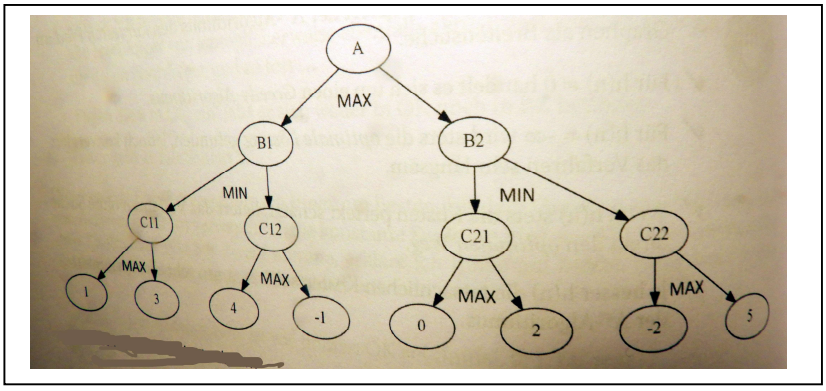


Figure 42.4: Variant tree

The starting position is A. The player at move wants to maximize the position evaluation (MAX level). Possible moves lead to the following positions B1 or B2, where it is the opponent's turn. All positions starting with C are now possible. At this level, the evaluation should be minimized (MIN level)

It is assumed that the opponent plays optimally. At the lowest level, I have only added the position value to the resulting nodes. Here it is the original player's turn again, so that when selecting the moves an attempt is made to maximize the evaluation (MAX level).

I will come back to the numbers in the nodes in the lowest level, which are not “expanded” any further, in a moment.

The min-max algorithm evaluates each node recursively from the lowest level to the top. C11, for example, is rated 3 because it is the maximum the player can get on his turn. C12 is rated 4 accordingly, but in B1 it is the minimizing player's turn. When selecting 3 (for CI 1) and 4 (for C12), he chooses the minimum, namely 3. In Figure 42.5, I have replaced the names of the nodes of the entire tree with the respective evaluations. The expected course of the train is marked by curved arrows.

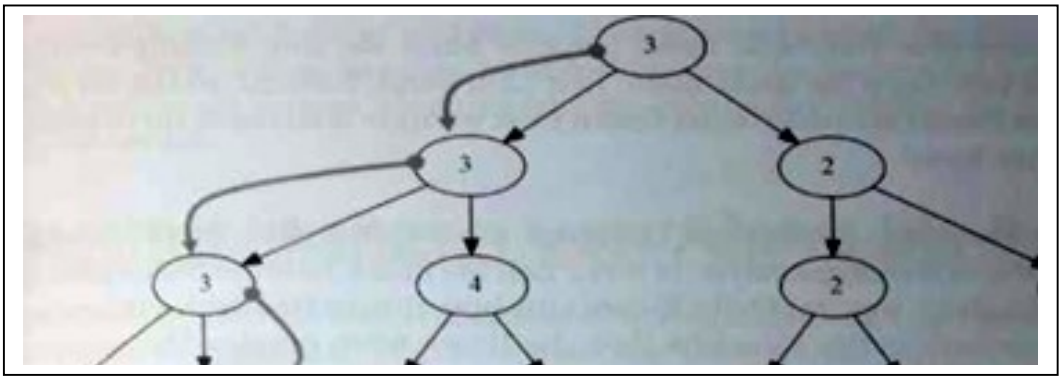


Figure 42.5 Completed evaluation tree

The mini-max method always finds the best move, provided that the complete position tree can be expanded. This is also referred to as a complete depth search. This is the case with simple games such as “Tic Tac Toe”. The numbers in the end nodes then contain the evaluation of a final position (win, draw, defeat). Write a program that uses the mini-max algorithm for this game and I guarantee that you will never beat it!

Unfortunately, this won't work for chess. On average, you have more than 30 sub-nodes to consider per position. With 40 moves - and some games last considerably longer - that would already be 30 to the 40th power; that's a number with 60 decimal places. For comparison: the number of atoms in our sun only has 57 digits...

So you have to stop your chess mini-max algorithm after a certain number of positions, even if these do not represent the end of the game. But how do you arrive at an evaluation?

You already know the solution! Namely via a .... Heuristics! There are many heuristics for position evaluation in chess, both simple and complicated:

- They count the remaining pieces.
- They add up the value of the pieces.
- You calculate the number of possible moves (the more, the better)
- You determine the “safety of the king”.
- ...

In the end, you use a heuristic that takes all these criteria into account and summarizes them into a single number.

However, there is a pitfall:

The more complex a heuristic, the more accurately it can evaluate a position. However, you need more time (effort, costs) for this, so that the max algorithm can use fewer positions for evaluation within a fixed period of time!

This is a classic trade-off, an interplay between a more precise and a deeper position analysis. The more time you invest in the depth of the position tree (search depth), the less time you have to apply a useful heuristic. Conversely, an elaborate heuristic costs you time, which you lack in order to examine the position tree in greater depth.

The American chess legend Robert (Bobby) Fischer was once asked how many moves he calculated in advance. His answer: “I don't calculate in advance at all. I win that way too.”

Translated for computer science: “In the interplay between depth search and heuristic position evaluation, I bet everything on heuristics. ”

You can take note of this with a smile. Using a heuristic means being creative, letting the mind prevail over matter. In contrast, depth-first search is a dull, repetitive process that is only fun for machines.

To be fair, I should note that today's chess grandmasters are convinced that any chess program with a depth search of ten moves in advance, provided with an extremely primitive heuristic position evaluation (mate and sum of material only) would probably beat any human opponent.

If you cannot expand the entire position tree, the crucial question arises: “When should the depth search be aborted and the resulting position heuristically evaluated?”

I can offer you a few possibilities:

- as soon as a predetermined number of moves is reached
- as soon as a final position (for example mate) is reached
- as soon as a predetermined time has elapsed
- as soon as the position is stable (i.e. when an exchange combination has been completed)
- as soon as there is (presumably) no more progress in the position evaluation
- ..

As you can see, it would be good if the variation tree could be pruned a little in order to extend the search in the graph. This is exactly what chess grandmasters do: they delve deeper into the position, but only analyze one or two important moves per node at a time. This requires intuition, which our machine method unfortunately does not have. Nevertheless, there is an extremely important optimization concept: pruning the tree of variants without causing a deterioration in the evaluation. How does this work? By simply cutting off branches that will certainly never be relevant for a specific move.

Pruning from alpha to beta

Consider Figure 42.4 again, assuming that our algorithm traverses the nodes in Pre-Order. This means that all sub-nodes of C11 are processed before it is C12's turn.

Need a little update on traversal? A quick look at chapter 36 will help you.

Next, the position with the rating “4” is analyzed. What does that mean? Since it is the maximizing player's move in C12, he would receive at least a rating of 4 for node C12, no matter what else comes next. Even if there were a thousand other moves from this node with a score greater or less than 4, they would not matter in the end!

This is because in B1 it is the minimizing player's turn and the evaluation of C11 is already less than 4 (namely 3). Since C12 has at least one value greater than or equal to 4, this branch will never be reached from B1! Take a look at Figure 42.5 and you will understand what I mean: The branch of C12 is ignored at the end. This is because you have come across a value (here “4”) that is worse than the value that the minimizing player receives in the worst case. We call this value, which is at least achievable for the maximizing player, a.

α (alpha) is the minimum value of the previous best (maximum) node for the maximizing player.

All remaining branches of C12 (in this case only the “-1”) are cut off. This is therefore a so-called a-cut-off.

An a-cut-off (alpha pruning) takes place at a maximizing level if a score for a position is higher than α. All other positions for the same parent node are cut off (i.e. ignored).

If you say “alpha”, you must also say “beta”, ß is the counterpart to αand contains the maximum value that the maximizing player will achieve. This of course corresponds to the minimum value that the minimizing player will receive.

ß (beta) is the maximum value of the previous best (minimum) node for the minimizing player.

As you can imagine, there is also a ß-cut-off:

A ß-cut-off (beta pruning) occurs at a minimizing level if a score can be obtained for a position that is smaller than ß. All other positions for the same parent node are cut off (i.e. ignored).

Exercise: Now I have a really difficult task for you: Can you find a ß-cut-off in Figure 42.4? Take a close look!

A beta-cut-off can only occur in a minimizing plane, namely when a position evaluation can be achieved that is smaller than the best that would otherwise happen to the minimizing player. The sub-nodes of B1 are eligible for this because the only value less than 3 has already been lost due to an a-cut-off. However, there is such a case with B2. First, C21 is calculated completely and assigned the value 2. However, 2 is less than 3 (the value of α and ß is the same after evaluation of B1, namely 3). This means that the entire branch of C22 is removed!

This seems crazy because there are both smaller and larger values below C22 compared to 3). However, the logical explanation should convince you:

- Suppose the positional evaluation of C22 were greater than 3. Then, of course, the minimizing player on the move would continue to follow path C21 and the possibilities of C22 are irrelevant.
- If, on the other hand, the position rating of C22 were less than 3, the minimizing player would think that was great, but the maximizing player would therefore avoid the entire branch (B2) because he already has the 3 on the other side.

As you can see, either way it is correct to remove the branch! Note that there could be other siblings of C21. They would all be ignored (or cut off) with the same reasoning, including their respective complete subtree. Ingenious, isn't it?

Figure 42.6 shows the truncated variant tree. Compare it with Figure 42.4! The pruning results in a significant increase in efficiency, although not a single relevant branch is missing.

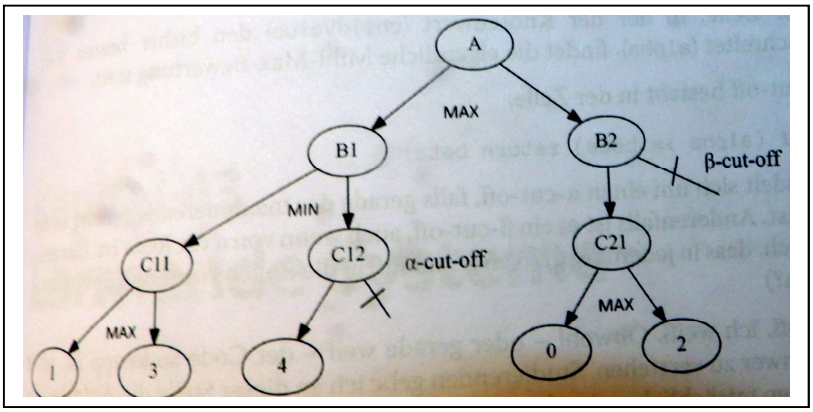


Figure 42.6: Pruned variant tree

I have deliberately refrained from presenting the code of the recursive mini-max algorithm in this section so far. It is more important that you first understand how it works. The alpha and beta pruning is now done with just a single line of code. Depending on the implementation, the role of the minimizing and maximizing player can be swapped by changing the sign at each level. In this case, a and ß must also swap their roles. Their signs are also reversed accordingly.

Your code for position evaluation (`eval`) could look like this:

```
int eval(int depth, int alpha, int beta) {
if (depth == 0 | | isLeaf()) return evaluatePosition();
PositionTree *child = expandChildren();
       while (child) {
              int childValue = -child->eval(depth-1, -beta, -alpha);
              if (childValue > alpha) alpha = childValue;
              if (alpha >= beta) return beta; child = child->nextSibling;
       }
return alpha; }
```

The maximum search depth is specified with depth and decremented in each subsequent recursive call. As soon as the value is zero, the depth search is aborted and the (possibly heuristic) position evaluation starts. This also applies in the event that no further moves are possible (`isleaf`)

The position evaluation is then of course correct (win, loss or draw) without heuristics.

Otherwise, the node is expanded (`expandChildren`) and the while loop runs through all siblings (`child->nextSibling`) in sequence (in pre-order).

The evaluation is called recursively, ß and α are swapped in their roles and negated (because the other player sets exactly the opposite priorities .

The actual mini-max evaluation takes place in the line in which the node value (`childValue`) exceeds the previous best value (`alpha`).

The cut-off is in the line:

```
if (alpha >= beta) return beta;
```

This is an α -cut-off if it is the maximizing player's turn. Otherwise it is a ß-cut-off, even if α is mentioned. (You remember that the roles of α and ß are reversed in every recursive subcall)?

I know, I know. Although - or precisely because - the code is so concise, it is very difficult to understand. At this point, I give students the task of actually implementing the procedure and including additional code to count the number of positions actually evaluated in the end with and without cut-off. You will be amazed at how many evaluations this single line saves you. The variant tree grows exponentially..

In this chapter:

- How to model arbitrary problems
- What you can expect from heuristics
- The basis on which our navigation systems work
- How to win easily with minimax algorithms

Chapter 43: Noisy systems

This chapter is about one of the most fascinating topics of all: How can you make a computer learn? I'll start by giving you an understanding of a few basic concepts before moving on to the actual algorithms. At the end, I'll even show you how to learn without a teacher ...

Machine learning

The idea of automating learning is an age-old vision of mankind. Countless science fiction stories revolve around the opportunities and risks of hyper-intelligent machines whose knowledge increases faster than that of their programmers. The drama of the novels, however, is that the inhuman mind usually ends up dying and the resolution is to switch off the machine - if that is still possible at all.

The fact is that we humans are extraordinary beings. Our minds enable us to learn from mistakes and apply existing experience to new situations. Learning is also common in the animal kingdom. Mice learn quite quickly to find their way around mazes when a reward beckons. They have to be presented with the same (or a very similar) situation again and again, and only the desired behavior is rewarded. Humans, on the other hand, learn much faster, even if children may see the 'L' in vocabulary learning differently.

But what exactly does learning mean? Psychologists usually answer that learning leads to a change in behavior. With machine learning, we have to be much more precise. A database storing music titles administering your music pieces behave in a different way, if you have saved a new piece. Nevertheless, you would NOT describe such a program as a learning system.

Learning is more, of a higher quality than simply saving data. In Figure 43.1 I have shown you how a learning system is basically structured.

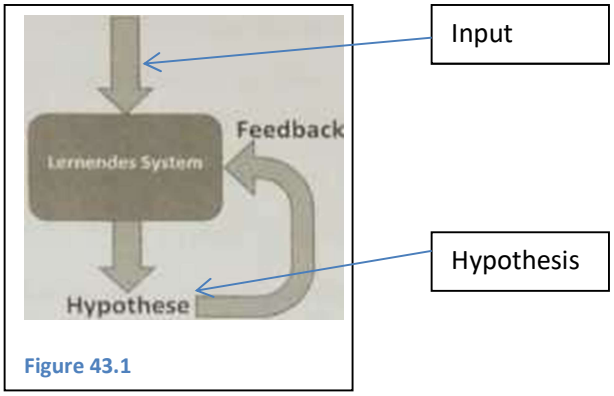


Figure 43.1

Input data is mandatory. If you do not feed the system with information, it cannot possibly learn. However, a learning system does not simply produce an output, but a hypothesis. There is more to this word than simple information. The hypothesis is an output whose truth has not yet been verified! It can therefore be wrong or right. Of course, you don't expect your music database to provide incorrect output. This means that database systems are out of the game when it comes to learning systems.

A program for predicting the weather, on the other hand, has the potential to be a learning system. The output is a forecast that can turn out to be right or wrong.

This is where it gets exciting. You can only speak of a learning system if this feedback is also used to improve future hypotheses. The feedback loop is of central importance, even if it remains within the system. Any program that does not care about this hypothesis is not a learning system.

Incidentally, this detail also applies to humans. Without feedback, no learning can take place. If I have no chance of finding out whether I am right or wrong in what I am doing, I will not learn anything. I go even further and claim that mistakes (or wrong hypotheses of a system) lead to real learning, while correct predictions are actually less important for learning progress.

Incidentally, this is an important reason why students should take advantage of the exam review after grading: Without feedback on where the failures were, effective learning does not take place! Although people also have subtle mechanisms for doubting their own “hypothesis formation” (an internal feedback loop), it is not uncommon for a review of one's own exam to lead to a few surprises

Inference without excuses

Figure 43 J. is of course still far too coarse, so I "zoom into" the "Learning system" and show th resukts in Figure 43.2-

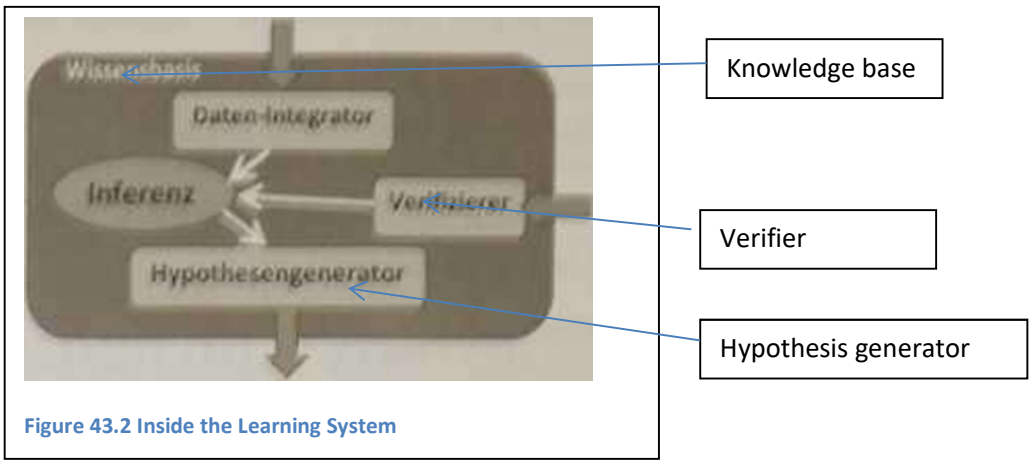


Figure 43.2 Inside the Learning System

The broad arrows at the edge come from the corresponding links in Figure 43.1. The data integrator has the task of processing incoming information and converting it into a form that the system can work with. The verifier receives the feedback from the last forecast and checks whether it confirms or refutes the hypothesis - this information is then available to the system just like the other inputs. The core of any learning system is inference.

Inference means (logical) reasoning (also known as deduction) as part of a learning process.

The concept of machine learning can also be examined from a different perspective in Chapter 11. There Figure 11.1 shows the "EVA" principle. Input -> Processing -> Output. As a matter of course, we assume that the input and the program are given to the computer and the output is generated from this. This is not the case with learning systems. You specify the input and the output and the computer calculates the "program", which is also called the "model" here! The wide arrows at the edge come from the corresponding links in Figure 43.1. The data integrator has the task of processing incoming information and converting it into a form that the system can work with. The verifier receives the feedback from the last forecast and checks whether it confirms or refutes the hypothesis - this information is then available to the system just like the other inputs. The core of any learning system is inference.

Inference means (logical) reasoning (also known as deduction) as part of a learning process.

The concept of machine learning can also be examined from a different perspective in Chapter 11 of your Dummies book, which looks at the general functioning of a computer. Figure 11.1 shows the "EVA" principle. Input -* Processing -* Output. As a matter of course, we assume that the input and the program are given to the computer and the output is generated from this. This is not the case with learning systems. You enter the input and the output and the computer calculates the "program", which is also called a "model" here!

Landing on the knowledge base

The purpose of inference is to generate new knowledge from existing knowledge. You can see in Figure 43.2, the entire system “floats” inside the knowledge base.

A knowledge base is a database whose content is prepared to infer information.

Please do not confuse the word inference, which is rarely used in everyday life with the much more frequently used term interference, which you encounter when waves are superimposed.

Let me give you an example of inference.

Suppose your knowledge base already contains the following information:

- If the traffic light is red, the car stops.

Your system receives the following new input:

- The traffic light is red.

Now the statement

- The car stops.

should now be inferred from the knowledge you now have. The statement itself was not previously part of the knowledge base.

inductive and deductive methods

The example is based on a propositional inference. In the course of this chapter, you will also become familiar with other inference methods: inductive and deductive methods

Basically, you should distinguish between two types of inference:

- Deductive inference starts from generally valid sentences and allowed rules and attempts to obtain true individual statements from them.
- Inductive inference - Conversely, inductive inference infers a generally valid law from individual experiences, which in turn allows the prediction of specific individual statements.

Only deductive inference is mathematically clean, whereas inductive inference is much more important and more common in the practice of learning systems. The reason for this lies in the typical fields of application of learning systems:

- Autonomous vehicles
- Mobile robots
- Voice and image recognition
- Weather forecast
- Stock market price prediction

- Lottery prediction

What all these areas have in common is that mathematically correct modeling is too costly, too complicated or simply not possible at all. The last catch is of course just a joke: by definition, random events cannot be predicted.

Noise in the data forest

There is another, no less important reason for the overriding importance of inductive inference: the world as we - and every machine - can perceive it is not one hundred percent consistent. The problem is not the world itself, but our possibilities of perception: Everything that we perceive with our senses and everything that measuring systems and computer sensors can pick up is subject to a known susceptibility to error. This is not so bad as long as we are aware of it. For example, our perception of what is warm and what is cold is very different.

Outsmarting the human perception of temperature

You are probably familiar with this experiment, but it doesn't hurt to repeat it here and there.

You fill three buckets with water. Bucket A contains cool water at 10 °C and is placed next to bucket B with a temperature of 30 °C. The water in the third bucket C, a little to the side, has a temperature of 20 °C. Now hold your left hand in bucket A for one minute and at the same time hold your right hand in bucket B. Logically, A will appear very cold and B very warm, that is clear.

However, it will be exciting if, after the minute, you put both hands in bucket C at the same time. Your left hand will feel pleasantly warm at the same temperature, while your right hand will feel unpleasantly cool!

The sensor technology of measuring systems is also subject to possible errors and defects. We say that the data contains noise, a certain amount of information that is distributed rather randomly and is therefore not useful for further processing steps.

Learning with a concept

After these rather theoretical preliminary considerations, let's move on to a concrete learning procedure. Concept learning is particularly suitable for this. The aim is to use a training set of examples, which may or may not correspond to a concept to be learned, to enable a general description of the properties of this very concept.

A concept consists of a set of properties that must have a certain value or are irrelevant. See for yourself:

You want your system to learn the concept of which birds belong to the species “blackbird”. Your training set consists of vectors whose components contain various properties of specific birds. In addition, you receive information about whether the respective animal is actually a blackbird.

In this simple example, only the following four properties are used:

- Songbird (yes, no)
- Plumage color (black, brown)
- Beak color (yellow, other)
- Size (small, large)

Your training set contains five vectors, each representing a combination of the above properties. Marked with an arrow -> to indicate whether it is a positive example (a blackbird) or a negative one (another bird species).

1. (yes, black, yellow, small) -> blackbird

2. (no, black, other, large) -> no blackbird

3. (yes, black, yellow, large) -> blackbird

4. (yes, brown, yellow, small) -> blackbird

5. (yes, black, other, large) -* no blackbird

The candidate elimination algorithm is used to arrive at a concept of blackbird based on a set of training examples.

It has two sets:

- The set G (General) contains the most general concepts, which include all positive examples and exclude all negative ones.
- The set S (Special) contains the most specific concepts that include all positive examples and exclude all negative ones.

In this case, a concept itself consists of a vector with four components, one for each property. However, an asterisk* is also possible. This means “this characteristic plays no role for the concept blackbird”.

G is initialized with the most general concept imaginable: G = {<*,*,*,*>}, where all properties are arbitrary.

Any example would be contained in G. Conversely, S starts with the impossible concept: S = {<-,-,-,->}, where no property is allowed and no bird is a blackbird.

Finally it starts. The algorithm treats all examples from the training set in sequence and adjusts the sets G and S after each step. The procedure distinguishes between positive examples (which represent blackbirds) and negative examples (which belong to a different species).

The sets G and S are successively adjusted after each presentation of an example. The elements from G tend to become more and more specialized, i.e. they include fewer and fewer candidates and thus restrict the concept of blackbirds more and more. The opposite is true for S. The concepts in it tend to become more and more general.

The following applies at all times: All the positive examples presented so far are contained in all the concepts of G and S. In contrast, not a single negative example fulfills a concept, neither from G nor from S.

It is best to show you separately how the candidate elimination algorithm proceeds with the individual examples, depending on whether they are positive or negative.

A positive example p is treated as follows:

- Delete all concepts from G that do not contain p.
- For each concept s in S that does not contain b...

  -delete s from S.

  -add to S all minimal extensions e of s that contain p and for which there is an element in G that is as general or more general than e.

  -delete all concepts in S that represent a generalization of another concept in S.

For a negative example n, the algorithm proceeds as follows:

- Delete all concepts from S that contain n.
- For each concept g in G that contains n...

  -delete g from G.

  -add to G all minimal specializations e of g that no longer contain n and for which there is an element in S that is just as special or even more special than e.

  -delete all concepts in G that represent a specialization of another concept in G.

This may sound a little complicated. Surely some things will become clear if you apply the algorithm to the training set from the last example:

The first attribute vector (yes, black, yellow, small) is a positive example because it describes a blackbird. G is not changed because this most general of all concepts contains any example, including the given one. The concept in S, on the other hand, must be extended so that it contains the example - just so. As a result, after introducing example 1, you get

G1 = G = { <*,*,*,*> }, S1 = { <yes, black, yellow, small> }

Now the second attribute vector (no, black, other, large) is added to the series. This is a negative example. This time, only G needs to be adapted because S excludes this example anyway:

G2 = { <yes,*,*,*>, <*,*,yellow,*>, <*,*,*,small> },

S2 = St = { <yes, black, yellow, small> }

Note that many possible concepts now appear in G, but all exclude the negative example. Do you notice that every attribute of the negative vector is simply negated? However, there is a problem with *black*. The concept <*, brown, *, * > is not a generalization of a concept from S, so it must not be included!

The next example is positive again (yes, black, yellow, large). It contradicts the last concept from G, which must therefore be eliminated. For S, you must find a minimal generalization of the existing concept. Can you do that? Here is the solution:

G3 = { <yes,*,*,*>, <*,*,yellow,*> }, S3 = { <yes, black, yellow, *> }

The penultimate example (yes, brown, yellow, small) is also positive. There is no change in G, as all existing concepts there also include the new example. However, the concept in S must be generalized:

G4 = G3 = { <yes,*,*,*>, <*,*,yellow,*> }, S4 = { <yes,*, yellow, *> }

When presenting the last example n = (yes, black, other, large), only G is adjusted. Minimal specializations of <yes,*,*,*> that do not contain n are <yes, brown,*,*>, <yes,*,yellow,*>, <yes,*,*,small>. However, only <yes,*,yellow,*> represents a generalization of a concept from S, so the other two are eliminated. However, <*,*,yellow,*> in G is more general than <yes,*,yellow,*>. The latter must therefore also be eliminated in accordance with the rule for negative examples. You get the final result:

G5 = { <*,*, yellow, *> }, S5 = { <yes,*, yellow, *> }

At the end, you obtain the set of all conceivable concepts from all elements in G and S. G contains the most general concepts, which include all positive examples and exclude all negative ones. The situation is similar in S, but the most specific valid concepts are stored there.

The set of all possible concepts are those from G, those from S and every generalization of a concept from S that is more specific than one from G. Although this does not occur in the above example, it does occur in practice.

In linguistic terms, our solution is

“A blackbird is a bird with a yellow beak”, which is the most general version, or ‘A blackbird is a songbird with a yellow beak’ for the most specific concept.

Which version you prefer depends on the use case. Let's assume you operate a waste sorting system and use the candidate elimination algorithm to filter out paper waste for recycling. Then you will probably use the more restrictive variant from S, because waste in paper is more harmful than vice versa. However, if the aim is to identify recyclable materials, you might use the more general version from G, because a human check is carried out anyway...

To find out whether a learning system is working (reasonably) correctly, you should present it with a test set. A test set is similar to a training set. However, the program must not initially rely on the assignment (positive or negative), but must first classify it itself according to the existing learning status. This is

called hypothesis generation. Feedback is then provided in the form of the correct classification and, if necessary, an adjustment of the level of knowledge, the actual learning. Once enough examples have been correctly classified, you can also use the system in practice.

Generally speaking, (inductive) learning systems work in two phases:

- In the ***training phase***, examples are presented. The system is adapted so that the examples would be classified correctly if their classification was not known.
- In the ***classification phase***, the learning system works productively. It uses the experience from the learning phase to (hopefully) classify unknown examples correctly.

The candidate elimination algorithm is a typical learning system. At the beginning, the knowledge base is empty and continues to fill up as the training phase progresses. The hypothesis formation does not take place explicitly here, even if you could (you could just as well classify the training examples presented beforehand). The concept assignment is the feedback. An adaptation of the sets G and S is learning in this sense. A pure classification, on the other hand, does not yet take feedback into account, but represents the formation of a hypothesis.

Only when feedback is received that the classification was incorrect does a (renewed) adaptation of the knowledge base take place, the actual learning. Surprisingly, positive feedback ("the concept assignment was correct") does not lead to a change in the knowledge base and therefore plays no role at all - for future learning success

Learning to decide with trees

Another variant of concept learning uses decision trees.

An introduction to decision trees as a data structure can be found in chapter 36.

The idea is to assign branches of a - in this case binary - tree to the individual properties of the examples presented. The thing is called a decision tree because it allows you to decide whether the presented example is a member of the concept or not based on as few questions as possible.

The example with the blackbird from the last section deserves further consideration. J

It is again about the question: "Is this bird a c?"

It is best to start by giving the examples a little more structure. I have listed them once in Table 43.1.

There is an obvious way to construct a decision tree from the given data. You start with the first attribute (songbird) and work your way from left to right. After the attribute evaluation, the next attribute comes next and so on. At the bottom, the leaves are labeled with the answer to the question of whether it is a blackbird ("YES" or "NO"). Figure 43.3 shows part of this tree.

There are two problems here. The first concerns the representation of the available information. For example, the feature vector "Songbird with black plumage and yellow beak" is a blackbird for both size variants (it is the first and third example from Table 43.1). The tree is therefore much too large. I could have added a "YES" directly after "songbird with black plumage and yellow beak".

The second one is even worse. While the "songbird with black plumage, non-yellow bill, large" is clearly not a blackbird ("NO") (last line in Table 43.1), I ask the question: What is a small animal with these characteristics? Blackbird or not? We do not have an example of this, which I have marked with the question marks.

Songbird	Plumage	Beak	Size	Blackbird?
Yes	black	yellow	small	YES
No	black	other	large	NO
Yes	black	yellow	large	YES
Yes	brown	yellow	small	YES
Yes	black	other	large	NO

Table 43.1: Examples and counter-examples of “blackbirds” as a concept

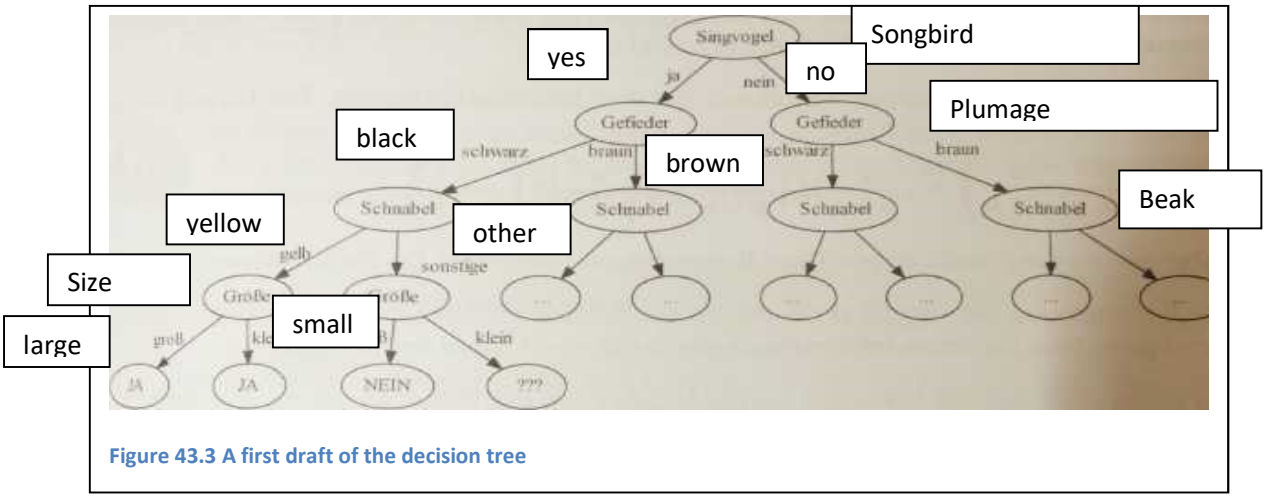


Figure 43.3 A first draft of the decision tree

The learning algorithm for the decision tree has two objectives:

- The tree should be minimal (in relation to the average path length of the training examples).
- The order of the queried attributes should be optimal.

Fortunately, there is a mathematically clean solution for this. The core idea of decision tree learning is as follows:

The decision tree is constructed in such a way that those attributes are queried first that yield the greatest information gain.

The only remaining task is to find the attribute that maximizes this information gain.

To do this, you should remember how the information content I of a character Z is calculated:

I(z) = ~ld(p(z))

Where p(Z) is the probability of Z occurring.

If you are not familiar with the calculation of the information content, just take a look at Chapter 51.

In order to determine the information content (of the application) of an attribute in the decision tree, we need to see what information content is subsequently found in the remaining examples.

Imagine that the application of an attribute splits the original set of examples into subsets that depend on the value of the attribute. For example, if you ask: “Is it a songbird?”, two new groups are created depending on the answer. Each of the groups in turn contains positive and negative examples of the original concept (“blackbird”).

Calculate the information content for each of the two groups. The formula for this is

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} ld\left(\frac{p}{p+n}\right) - \frac{n}{p+n} ld\left(\frac{n}{p+n}\right)$$

Where “p” is the number of positive examples and “n” stands for the negative ones.

At the end, of course, you have to summarize the values of the resulting groups by weighting each information content with the number of elements.

Does that sound exhausting? Maybe it's easier than you think. Try it out now!

The following values result for our blackbird from Table 43.1.

The attribute “songbird” splits the original set of five examples into two groups. The first group contains examples 1, 3, 4, 5, while only example 2 is ***not*** a songbird.

Under the first group you will find three positive (p = 3) and one negative example (n = 1)* You can recognize this by the “YES” or “NO” in the last column.

This is a dangerous source of error. The positive (p) and negative (n) example sets always refer to the basic concept within the group division, in the current case therefore to the question of how many blackbirds there are among the songbirds (not how many songbirds there are among all examples, which would be p + n).

Therefore, the information content of the first group results in:

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{3}{3+1} ld\left(\frac{3}{3+1}\right) - \frac{1}{3+1} ld\left(\frac{1}{3+1}\right)$$
$$= -\frac{3}{4} ld\left(\frac{3}{4}\right) - \frac{1}{4} ld\left(\frac{1}{4}\right)$$

Following the laws of logarithms (a quick look at Chapter 49 will also help here), the following simplification results:

$$I\left(\frac{3}{4},\frac{1}{4}\right) = -\frac{3}{4}ld(3) + \frac{3}{4}ld(4) - \frac{1}{4}ld(1) + \frac{1}{4}ld(4)$$

since ld(4)=2 and ld(1) = 0 you will get:

$$I\left(\frac{3}{4},\frac{1}{4}\right) = -ld(3) + ld(4) \approx 0,415$$

The second group only consists of a single example. In this case, it does not matter whether it is positive or negative, see for yourself:

$$I\left(\frac{0}{0+1},\frac{1}{0+1}\right) = I(0,1) = -ld(1) = 0$$

There is a little mathematical trick in this formula: What is zero times the logarithm of zero? It can be shown with mathematical finesse (see box) that this limit is really zero.

Borderline considerations...

It is a misconception to think that only linear algebra is relevant for computer scientists in the field of mathematics, but not analysis, which deals with calculus and limit value calculation. The story with the limit value for "times logarithm dualis of" proves the complete opposite.

Here is the short form of the solution. In mathematical terms, it is about the value of the formula:

$$\lim_{\varepsilon \to 0} \varepsilon \cdot ld(\varepsilon) = \lim_{x \to \infty} \frac{1}{x} \cdot ld\left(\frac{1}{x}\right)$$

The small letter on the left, which looks like a capital "E" in cursive writing, is the Greek (small) epsilon. It is typically used for positive numbers that are "infinitely" small, almost zero. This is indicated by the "lim" for "limes" (Latin for "limit") in the formula. We are therefore looking for the limit value towards zero of a tiny number epsilon, which is to be multiplied by its logarithm dualis - which then actually tends towards (minus) "infinity". The question could be formulated boldly as follows: "What results in zero times infinity?"

On the right-hand side of the equals sign, I have replaced the epsilon, which tends to zero, with an x, which now tends to (plus) infinity. "Epsilon towards zero" is analytically equivalent to 'one through x towards infinity'. You can also write this product as a fraction:

$$\lim_{x \to \infty} \frac{1}{x} \cdot ld\left(\frac{1}{x}\right) = \lim_{x \to \infty} \frac{ld\left(\frac{1}{x}\right)}{x}$$

According to the third law of logarithms, the Jd of 1/x is the same as the negated logarithmus dualis of x. I put the minus right before the whole fraction:

$$\lim_{x\to\infty} \frac{1}{x} \cdot ld\left(\frac{1}{x}\right) = \lim_{x\to\infty} \frac{ld\left(\frac{1}{x}\right)}{x}$$

Now it gets tricky. According to the mathematicians Johann Bernoulli and Guillaume de L'Höspital, you can replace the numerator and denominator functions with their respective derivatives at this point. In the case of the logarithm, the result is "one by x" multiplied by the constant logarithm naturalis of 2 (ln(2)), while the denominator becomes 1. You can then leave it out:

$$\lim_{x\to\infty} -\frac{ld(x)}{x} = \lim_{x\to\infty} -\frac{1}{x \cdot \ln(2)}$$

Now a fraction whose denominator gets bigger and bigger as x increases will itself end up being zero:

$$\lim_{x\to\infty} -\frac{1}{x \cdot \ln(2)} = 0$$

Which was to be shown.

Further fun with logarithms, a special field of analysis, can be found in chapter 51. Of course, I am in no way implying that linear algebra is unimportant for computer scientists, quite the opposite...

The information content of a set in which all examples are of the same type is zero. This is also logical: with such a selection, you need exactly zero questions to classify the set...

$$I(\text{Singvogel}) = \frac{4}{5} I\left(\frac{3}{4}, \frac{1}{4}\right) + \frac{1}{5} I(0,1) \approx 0,332$$

songbird

Overall, both information contents must be weighted and added together to obtain the total information content of the "songbird" attribute. Four out of five examples were songbirds, only one could not sing:

$$I(\text{Gefieder}) = \frac{4}{5} I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{5} I(1,0) = 0,$$

plumage

What does this tell us? This value alone does not really help you. You need to do this work for all the other attributes as well

$$I\left(\frac{1}{2}, \frac{1}{2}\right) = \frac{1}{2} ld(2) + \frac{1}{2} ld(2) = \frac{1}{2} + \frac{1}{2} = 1$$

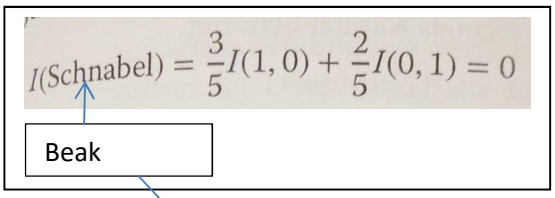
If an attribute produces just as many positive as negative examples, as in the case of black plumage, this results in an information content of

This is the maximum you can get. However, this is exactly what you are NOT aiming for

The question about the plumage produces a higher information content in the residual menu than the question about the songbird. However, you do not want to maximize the information content, but rather the information gain.

The information gain is maximized for the attribute for which the information content of the remaining set is minimal.

Accordingly, it would not be a wise idea to ask for the plumage first. However, you still have two other attributes in the program



As soon as you ask about the beak, the resulting groups contain a remaining information content of zero. All examples with a yellow beak are positive, all others are negative. Your decision tree would therefore only have to ask for exactly this

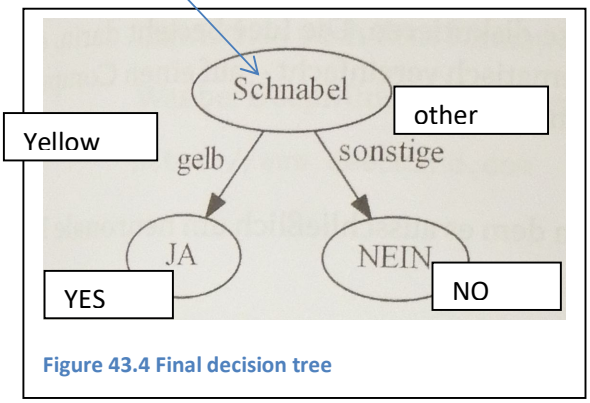
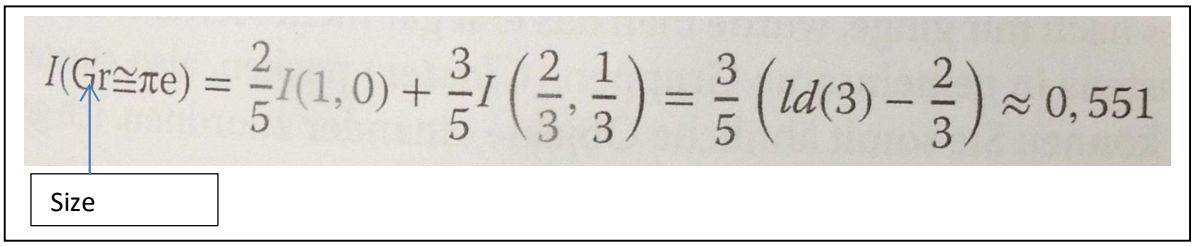


Figure 43.4 Final decision tree

The information content cannot be less than zero, so you do not necessarily have to ask about the size. However, if - for the sake of completeness - you prefer to calculate this value as well, voilä:



The example raises a question. Perhaps you are not yet aware of what you do if none of the attributes leaves information content of zero? In this case, you simply select the one with the lowest value and generate the top branch of your decision tree, very similar to Figure 43.4. However, you are then - in general - not yet finished. Then apply the algorithm recursively to each child node. On the one hand, the sample set is reduced because you only consider those that have the same feature in relation to the application of the last attribute. On the other hand, the number of attributes is also reduced because you do not need to look at an attribute that has already been used once again. Finally, if the information content remains zero, you are finished and the decision tree is in full bloom.

At the end of this section, I would like to draw your attention to something else. Compare the result of the candidate elimination algorithm with decision tree learning. They are not identical! At least not exactly. As you can see, the decision tree only contains the most general concept from G. In this respect, you can also generate a decision tree from the concept learning, but not vice versa.

Learning without a teacher

There are of course a number of other learning algorithms, such as stochastic approaches. There are also expert systems that represent the knowledge of human professionals. This is a tricky matter that requires its own chapter

- Chapter 44 is all about expert systems in general and “case-based reasoning” in particular.

Of course, we also need to discuss neural networks. The idea is to simply transfer the functioning of our brain - in mathematically simplified terms - to a computer. We also need a separate chapter for this.

- You can look forward to chapter 45, which deals exclusively with neural networks.

All of these methods mentioned belong to the same group of supervised learning algorithms. This means that you already have labeled training data available for your algorithms. This contrasts with unsupervised learning, where the labels are missing. What is that supposed to do? How is learning supposed to be possible? In the strict sense, of course, it's not possible, and if it were up to me, it wouldn't even be called “learning”. At best, you can pack the data into common groups (clusters). Without knowing the labels, you can assign similar objects to each other. I admit, that's kind of cool.

Another group of methods is still missing, which is a bit between supervised and unsupervised learning. This involves “learning without a teacher”.

Learning without a teacher happens without human intervention, it has to happen on its own, so to speak. For example, in water, you can - theoretically - learn to swim without a teacher. The activity as such will give you feedback on whether you are making progress or whether you are on the wrong track. However, I really can't recommend this, the risk of drowning is incredibly high and the risk of dying would only be increased by trying to learn to fly without an instructor ...

However, there are many areas where you can actually teach a computer (which can also be in a mobile robot) to learn without an instructor. In English, the term “reinforcement learning” has been coined for this.

Just think of the search for an exit from the labyrinth. The environment will tell you when you have found the exit. In between, you can learn to find the exit without a teacher, or chess or any other type of board game. The rules are sufficient, and your program can work out for itself which moves were good and which were bad - depending on whether you win or lose.

Thanks in part to the AlphaZero program from the London-based company DeepMind, which is now part of Google, breathtaking progress has been made in the field of reinforcement learning over the past few years. For example, AlphaZero trains itself to become better at chess. Unlike classic engines, it does not

use any opening books or grandmaster knowledge. DeepMind had previously developed AlphaGo, the first program to beat the human Go world champion. But DeepMind has much more up its sleeve. AlphaFold is able to predict 3D models of protein structures like no other program before it. This is expected to lead to revolutionary breakthroughs in pharmacy and medicine. DeepMind has now also turned its attention to climate change.

You can find out what else is currently going on at DeepMind on the homepage:

```
https: //www. deepmind. com
```

In this chapter:

- Get to know Prolog as a logical programming language
- Experience application examples of expert systems
- Closing case bases
- Planning and predicting with artificial intelligence

Chapter 44 Expert systems for professionals

Expert systems represent the knowledge of human experts. This is ultimately what this chapter is all about. However, we start with an old, impressive programming language that allows you to draw logical conclusions. After that a large part of this chapter then deals with a special procedure for storing expert knowledge, “case-based reasoning”. I will also show you what you can do with it.

Prologue

There is a separate class of programming languages for logical reasoning. The most famous representative is called Prolog, which stands for “Programmation en Logique”. This language was developed back in the 1970s by French computer scientist Alain Colmerauer.

- The programming language Prolog is not pronounced like the introduction to a poem or a novel, but the second syllable corresponds to the log of “login”.

The idea is to formulate the requirements of a task in the form of logical statements and to use the Prolog interpreter to draw the right conclusions from them. The more information you put into the system at the beginning, the more exciting the logical conclusions will be.

Let's start with the simple statement “Adam is a man”. This looks in Prolog:

```
man(adam).
```

Note the period at the end of each command. Accordingly - we are practicing equality - Eve is a woman:

```
woman(eve).
```

Constants in Prolog always start with lowercase letters, while variables start with the underscore “_” or an uppercase letter.

It only gets exciting with implications. For example, if you want to say that C follows from A and B, i.e. A A ^ B -> C, write it down in Prolog in this form:

```
C :- A,B.
```

As you can see, the result of the implication is at the beginning. This is followed by the somewhat difficult to read “:-”, which replaces our operator. The conditions follow at the end. AND links are achieved using commas, while OR links are connected with a semicolon.

Take a look at the following example:

```
human(X) :-  woman(X);man(X).
```

Can you guess its meaning? For the first time we are working with a variable, namely “X”. The connection between the woman and the man is a semicolon. Accordingly, you read this prologue expression as: “X is a human, if X is a woman or X is a man.” Other I conditions could also characterize a human being, so the statement is not exclusive (don't worry!).

Now Prolog is already able to conclude a first statement of its own:

```
human(eve).
```

If you want to test out Prolog, I recommend the GNU Prolog implementation for all common platforms (http: //www. gProlog. org). The example above will then look like Figure 44.1.

The header appears automatically. The prompt symbol is “`| ?-`”. You can only enter your own commands after this. To do this, first switch to user mode using `[ user ]`. Alternatively, you can also enter your statements in a separate file. Use the key combination <Strg> + <D> to exit the mode, which causes the Prolog interpreter to issue the statement `user compiled ...`

Then enter one statement at a time and the system will tell you whether the story is true or false.

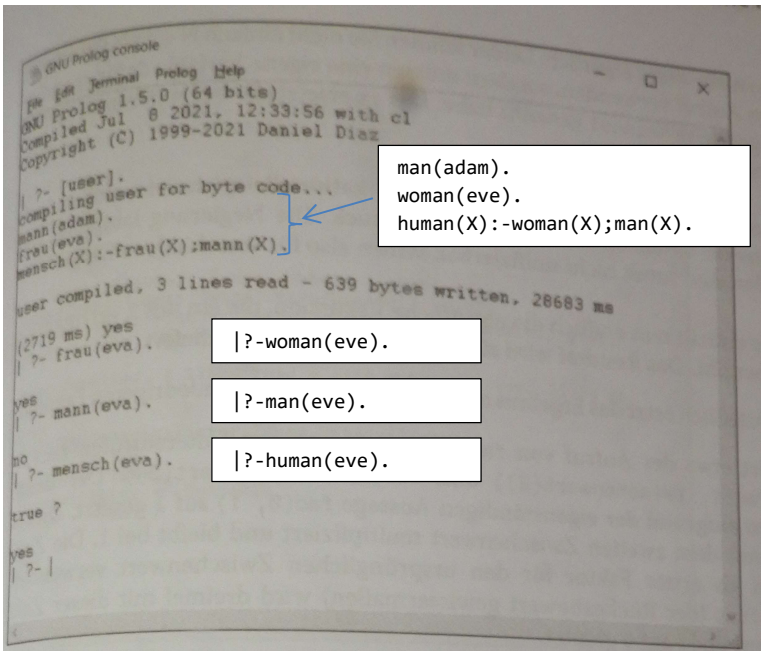
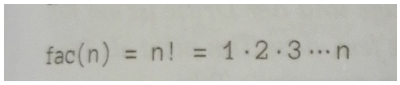
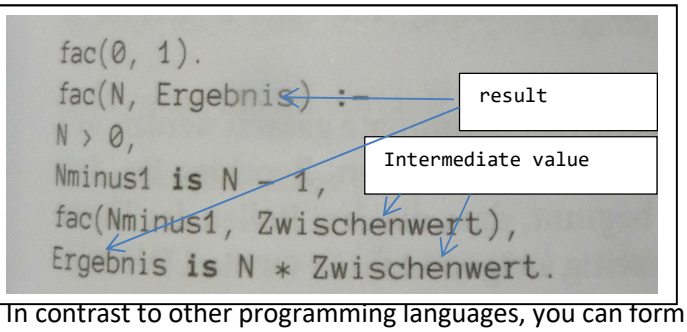


Figure 44.1 GNU-Prolog Console

Of course, you can also do more complicated things with Prolog, such as recursive definitions. A classic example is the factorial:



It is best to enter the result as the second parameter in Prolog:



In contrast to other programming languages, you can formulate the termination condition of the recursion as an independent statement: `fac(0, 1)` . means that the factorial of zero is one, i.e. `0! = 1`.

The actual function can be found in the following rule. As you already know, variables are capitalized in Prolog, and this also applies to the arguments N, the number whose factorial you are interested in, and `result`, which stores the return value. The first condition is `N > 0`, which is very important because otherwise it would not be ensured that the first line `fac(0, 1)` , really applies to the factorial of zero. Because the function body is not yet finished, a comma follows.

The next line is a little annoying. Unfortunately, you can't just use `N -1` as the para for the recursive call, you have to spend a separate variable for it, which I have sensibly called `Nminus1`. This `is` replaces the "=" assignment that is common in other languages

The operator = means unification in Prolog. For example X = Y both variables stand for the same expression. The negation is `\=` . Variables are therefore not unifiable, i.e. they stand for different things.

The penultimate line finally contains the actual recursion, which passes an `N` reduced by 1 as an argument. The result is returned as an intermediate value.

Finally, the last line sets the result to the `N`-fold of the next lower factorial

In detail, the call of `fac(3, Result)` leads to the subcalls `fac(2 intermediate value), fac(l, intermediate value(2))` and `fac(0, intermediate value (3))`. The third intermediate value is set to 1 due to the independent statement `fac(0, 1)` to 1.This result (1) is multiplied by the second intermediate value and remains at 1. The number "2", on the other hand, is used as the first factor for the original intermediate value. The variable `result` (the return value, so to speak) is multiplied three times by this number and set to 6 at the end.

As another example, I would like to show you how to solve a small criminal case with the help of Prolog.

Three men named Tim, Tobi and Benjamin are suspected of the crime. One of them must be the perpetrator, a second a witness and the third is not involved in the crime. However, he is not an innocent bystander either, just like Tobi.

The implementation in Prolog and the resolution of the case can be found in Figure 44.2.

The names of the persons must be placed in single quotation marks, otherwise they would be interpreted as variables. Note that the rule for solving the case begins by mutually excluding the three roles of perpetrator, witness and bystander. Such facts are so obvious to a person that they are often used unconsciously and forgotten as explicit input.

The resolution at the end describes the explicit roles of all three persons.

Expert knowledge

Imagine you were to store all conceivable facts in Prolog. This could be banal things like names or addresses that you find in the phone book, or more personal data, such as your birthday, hobbies and much more. In principle, this would look like

```
human( 'Müller', 'Helene', birthday(30,11,1966)).
```

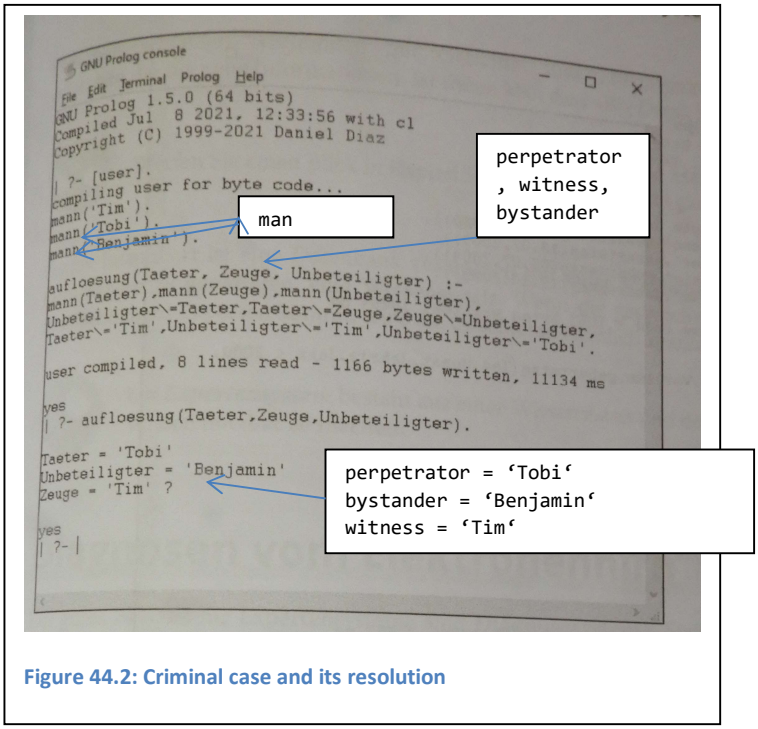


Figure 44.2: Criminal case and its resolution

```
human ( 'Meier', 'Werner', birthday(13,07,2009)).
human (' Becker',' Sophie', birthday (02,07,2004)).
human ( 'Fries', 'Werner' , birthday (19,10,2003)).
```

You could now very simply search in this data, for example,

```
human (surname, first name, birthday (day, month, year)),year < 2000.
```

returns the person who was born in the previous century.

It is a little more convenient with `findall`.

```
findall  (X, human (X, 'Werner', _), result).
```

stores the surnames of all “Werner”s from the knowledge base in the Result list.

The `human` corresponds to the table of a database. You could now split another database and join the two:

```
month (11, 'November').
month (07, 'July').
month (10, 'October').
human (last name, first name, birthday (day, X, year)),monat(X, month).
```

Figure 44.3 shows the result of this and all previous queries. The variable X is used to create a **join** between the tables for man and month.

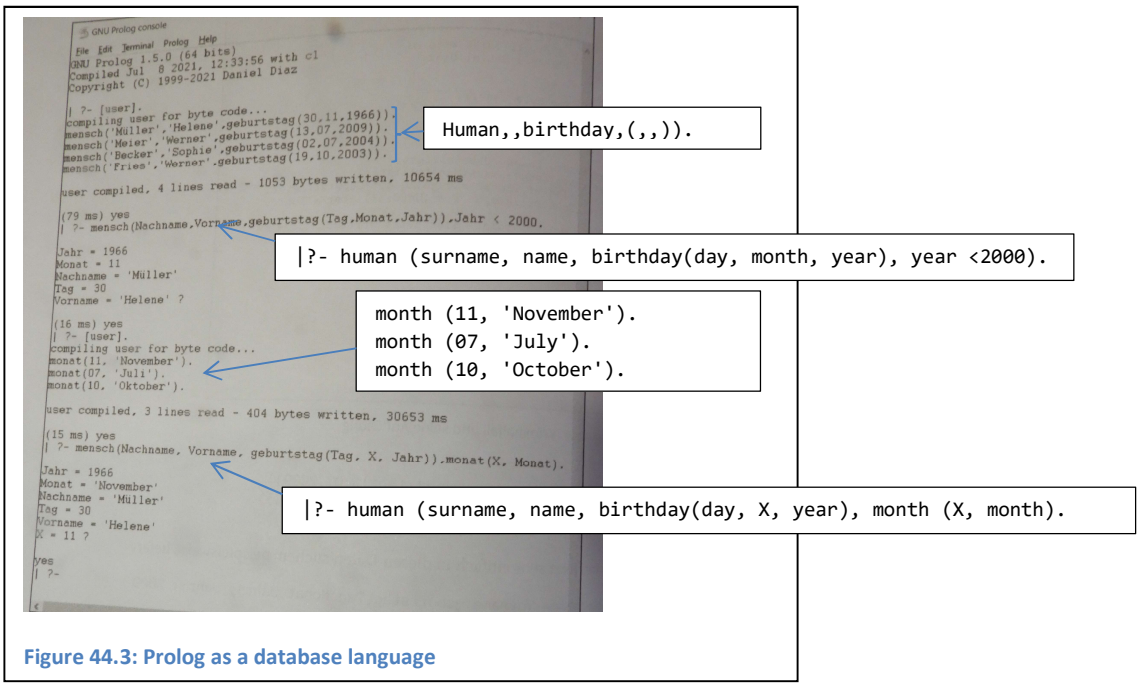


Figure 44.3: Prolog as a database language

You know all this from SQL? That's right, such tasks can easily be solved with a database language.

You can find a crash course on SQL in chapter 40.

What actually happens when you connect the logical conclusions in Prolog with its database properties?

You could save the fact that Werner Meier is an uncle of Sophie Becker and use a logical step to infer the family relationships of all other persons.

Combined with facial and event recognition algorithms, new information is generated. Do you realize that “big data” has long been a reality?

Take a look at Chapter 50 if you don't believe me ...

Positively formulated, a database together with a possibility to generate new information from it is a knowledge base. When the stored information comes from (human) experts, the story gets pretty exciting.

An expert system consists of a knowledge base and the possibility of making this knowledge usable.

Diagnoses from the electronic brain

Diagnostic systems are a good example of expert systems. Imagine that there are only a few experts in the world for very specific diseases. If you succeed in making their knowledge available electronically to all doctors and clinics, you will have taken an important step forward. In the search for a specific disease, all possible symptoms are recorded, and from this you deduce a diagnosis. The information required for the logical conclusions comes from the knowledge base.

The problem here is that you cannot, of course, expect the exact case you have in front of you to already exist in the database. Instead, you need a case that is as similar as possible.

In order to draw a meaningful conclusion as to which type of symptoms are relevant for the respective clinical picture, you need to distinguish between different categories of knowledge:

- Factual knowledge
- Domain knowledge
- Metaknowledge

Factual knowledge corresponds to the input of human experts based on their concrete experience. Here, for example, a list of the cases already recorded with symptoms and diagnoses comes into question.

Domain knowledge contains important correlations that are relevant to the respective field - for example, medical diagnosis. This includes, for example, a tense abdominal wall in the case of appendicitis or the typical nausea in the case of concussion.

Just imagine that the domain knowledge comes from a textbook, while the factual knowledge has to do with concrete experiences of the respective doctors.

There is also the meta-knowledge, the knowledge about the knowledge itself. In the context of expert systems, we understand this to mean implementation specifications. For example, you could build up your knowledge base in the form of prolog statements. However, by choosing the programming language, you automatically favor certain inputs over others.

The unavoidable mistae, which you makek as soon as you create a knowledge base and which stems from meta-knowledge is called bias.

A knowledge base is made up of different forms of knowledge. Factual knowledge contains concrete experience, domain knowledge contributes generally known relationships and meta-knowledge is applied through the IT implementation of the overall system. The associated fundamental influence on the creation of knowledge is called bias.

I would like to pick out one of the many possible implementations for diagnostic systems and present it to you in more detail:

Case-Based Reasoning.

The idea of case-based reasoning (CBR) is to store previous (medical) reference cases in the knowledge base. You can imagine a case as a pair of vectors. The first vector is the symptom vector, the second is the diagnosis vector.

In somewhat abstract terms, a case generally stands for the description of a problem (symptoms), combined with the associated solution (diagnosis).

To be able to process the symptoms mathematically, you need to convert their values into numbers. For example, you could use the following coding for “fever”:

0: none

1: slight

2: medium

3: high

4: very high

Or you can simply convert the concrete specification of Celsius into a numerical value between 0 and 10. There are no limits to your imagination. However, you should use numerical scaling in order to be able to draw real case-based conclusions.

At the end of the day, your symptom vector will consist of numbers. Each component represents a specific symptom.

If a symptom is unknown (or simply not recorded), select a special character that is not a number instead, for example ¥

The diagnosis vector, on the other hand, consists only of truth values. A patient could suffer from several diseases at the same time, in which case several components would be 1 (true). However, if all values are 0 (false), no diagnosis is made at all.

In this very simple example, there are only four different types of symptoms and three possible diagnoses. Because words are smoke and mirrors, I will limit myself to purely numerical information. The case base F, the special knowledge base in the CBR, consists of only three cases:

$$F = \left\{ \left(\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right), \left(\begin{pmatrix} 6 \\ 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right), \left(\begin{pmatrix} 5 \\ 2 \\ 3 \\ 7 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right) \right\}$$

Each case is a symptom vector linked to the corresponding diagnosis vector. For example, a new patient could have the following symptoms:

$$\begin{pmatrix} 0 \\ 2 \\ 3 \\ 1 \end{pmatrix}$$

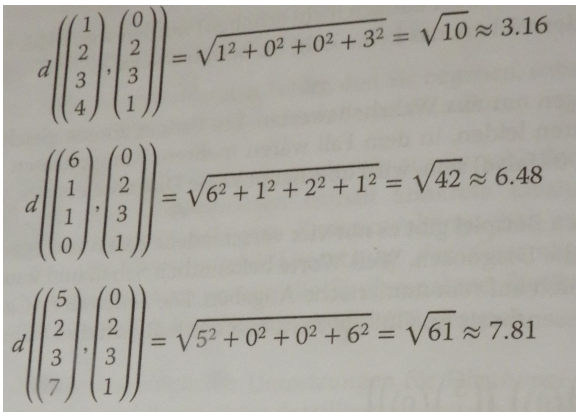
What is his diagnosis?

Okay, just a moment. Before I explain in detail how you arrive at your diagnosis, let's first take a look at the overall artwork.

The principle of case-based reasoning consists of four steps:

- Retrieve: Determine the case C most similar to the new symptom vector S from the case base.
- Reuse: Apply the diagnosis D from C to S.
- Revise: Investigate whether D is actually an appropriate diagnosis for S.
- Retain: Save (S, D) in the case base if a new reference case is found here.

The most exciting point is the similarity analysis. Various measures (metrics) are conceivable here. For example, consider each symptom vector simply as a position vector in four-dimensional space. The distance between these spatial points (Euclidean norm) then simply corresponds to the similarity: The closer, the more similar. The differences of the corresponding components are squared, and at the end the square root is taken from the sum. Just give it a try:

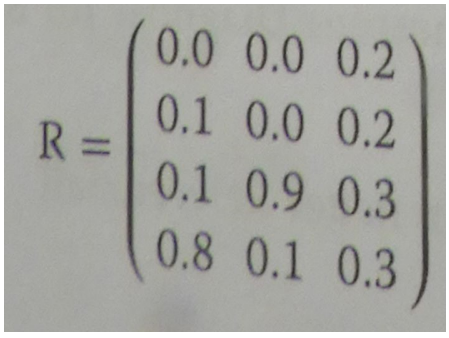


For numerical reasons, you can also omit the root extraction at the end. The smallest root naturally also corresponds to the smallest value under the root.

Therefore, the first case would be the most similar!

The calculation becomes more complicated when taking unknown symptoms (¥) into account. A simple Euclidean calculation then no longer helps. In this case, special case distinctions are made, depending on which side (new system vector S or component of a case C of the case base).

Now another thought comes into play. Are all symptoms really to be evaluated in the same way for every disease? If so, you would have a hard time recommending case-based reasoning to practicing physicians. Rather, you have to weigh the individual symptoms. Learning in CBR takes place by adjusting these weights. For example, it turns out that the symptom of toothache does not play a major role in the diagnosis of myocardial infarction, even though this may occur in individual cases. On the other hand, severe chest pain, possibly with severe pain in the upper arm, could be highly relevant.



If you combine the individual weights of the different symptoms with the possible diagnoses, you get a matrix, which is also known as the relevance matrix R. In our example, this would be something like:

Each row indicates how important the respective symptom is, the column stands for one of the three possible diagnoses.

Note that the column sum must be normalized to 1, which corresponds to the relevance of all symptoms per diagnosis.

With this factor, the similarity calculation looks much more complicated, but more realistic: Once again, all three cases in the case base are considered. This time, however, you must also take the target diagnosis into account.

The similarity to the first case (diagnosis 1) is calculated using the first row of the relevance matrix. Please note that we are looking for the smallest distance at the end. Therefore, you must first distinguish for each attribute whether it is fulfilled or contradictory. For example, if the case from the case base requires a high value for high fever in the lowest component, but the new symptom vector has zero fever at this point, the weight in R would not be taken into account if it were naively multiplied by zero.
You therefore use a somewhat more sophisticated formula to calculate the similarity (sim). You first add up all the weighted fulfilled attributes to the value E, while the contradictory attributes are added up to W. You obtain the similarity. You then obtain the similarity from the quotient:

$$sim(C,S) = \frac{E}{E+W}$$

If there are no contradictory attributes, all are fulfilled and the value is “1”. However, if the contradictory attributes are in the majority, the similarity decreases until the break finally results in zero if no attributes are fulfilled at all.

The similarity between a (new) symptom vector S and a case in the case base C is normalized between a maximum of 1 (highest similarity) and a minimum of 0 (no similarity).

If you want to be on the safe side, you can add additional factors to E and W to control the similarity. However, in order to keep an overview, I will spare you such trivia. The best way to do this is to use the example again.
First you have to define when an attribute is fulfilled and when it is not. Is 38 degrees already a fever? Certainly not a high one. Is a blood sugar level of 115mg/dl already high for a sober person? If it is a child, perhaps yes.

These questions must be answered by the expert whose knowledge is required when creating the initial cases in the knowledge base. For our example, we simply make the definitions ourselves: All attributes assume values between 0 and 8, using the following formula for direct comparison:

$$1 - \frac{|A_F - A_S|}{8}$$

As is the attribute value of the symptom vector, while AF is the corresponding value from the case base. If both are the same, you determine 1 - 0 = 1, the degree of fulfillment. Otherwise, the difference could be at most 8, in which case the formula calculates a 0.

For numerical reasons, the denominator is often increased by one, but this is not important for us here. We could agree that the difference in the attribute values should be at most 2, so that the formula at least reaches the value:

$$1 - \frac{2}{8} = 0.75$$

.. to speak of a fulfilled attribute, otherwise it is a contradictory one.

Comparing the first two vectors $\begin{pmatrix}1\\2\\3\\4\end{pmatrix}$ and $\begin{pmatrix}0\\2\\3\\1\end{pmatrix}$ results in the following value for the top attribute

$$1 - \frac{|1-0|}{8} = 0.875$$

the two middle ones are identical, resulting in the value 1 and for the lowest component you obtain

$$1 - \frac{|4-1|}{8} = 0.625.$$

Only the last attribute is contradictory, while the other three are fulfilled. You determine the following values for H and W, taking into account the relevance matrix - the first column is to be considered here because the case from the case base has the highest diagnosis:

$$E = 0.0 \cdot 0.875 + 0.1 \cdot 1 + 0.1 \cdot 1 = 0.2$$

Although the top three attributes are fulfilled, E is not particularly large because the corresponding weights for diagnosis 1, which is present in this case, are fairly unimportant.
For W the result is

$$W = 0.8 \cdot (1 - 0.625) = 0.125$$

Do not forget that for all contradictory attributes the determined value must be subtracted from 1, because very small values indicate particular contradictions.
As similarity you get

$$sim\left(\begin{pmatrix}1\\2\\3\\4\end{pmatrix}, \begin{pmatrix}0\\2\\3\\1\end{pmatrix}\right) = \frac{E}{E + W} = \frac{0.2}{0.2 + 0.125} \approx 0.615$$

I will now show you the same game for the other two cases from the case base

For $\begin{pmatrix}6\\1\\1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\2\\3\\1\end{pmatrix}$

result in the attributes from top to bottom in sequence: $1-\frac{|6-0|}{8}=0.25$ (contradictory)

$1-\frac{|1-2|}{8}=0.875$ (fulfilled)

$1-\frac{|1-3|}{8}=0.75$ (just so fulfilled) and finally

$1-\frac{|0-1|}{8}=0.875$ (fulfilled).

When calculating E and W, make sure that you always select the correct column from the relevance matrix, depending on the diagnosis of the respective case. This is the second column:

$$W = 0.0 \cdot (1 - 0.25) = 0.0$$

If W is zero, the overall similarity must always be maximum, make sure of this:

$$E = 0.0 \cdot 0.875 + 0.9 \cdot 0.75 + 0.1 \cdot 0.875 = 0.7625$$

and thus

$$sim\left(\begin{pmatrix}6\\1\\1\\0\end{pmatrix},\begin{pmatrix}0\\2\\3\\1\end{pmatrix}\right)=\frac{0.7625}{0.7625+0.0}=1$$

In the third case, you get $\begin{pmatrix}5\\2\\3\\7\end{pmatrix}$ for $\begin{pmatrix}0\\2\\3\\1\end{pmatrix}$ $1-\frac{|5-0|}{8}=0.375$ and (contradictory)

$1-\frac{|2-2|}{8}=1$ (fulfilled)

$1-\frac{|3-3|}{8}=1$ (fulfilled)

$1-\frac{|7-1|}{8}=0.25$ and finally (contradictory)

This gives you the following for E and W.

$$E = 0.2 \cdot 1 + 0.3 \cdot 1 = 0.5$$

$$W = 0.2 \cdot (1 - 0.375) + 0.3 \cdot (1 - 0.25) = 0.35$$

In total, this results in :

$$sim\left(\begin{pmatrix}5\\2\\3\\7\end{pmatrix},\begin{pmatrix}0\\2\\3\\1\end{pmatrix}\right)=\frac{0.5}{0.5+0.35}\approx 0.588$$

The most similar case is the second, which is why it is used in the “retrieve phase” of case closure.

But how does learning take place? In learning mode, the aim is to adjust the values of the weights, whose initial state may be estimated by experts, through empirical investigation. To do this, you draw symptom vectors of already diagnosed cases and pretend that the diagnosis is unknown to you. There are two possibilities for the resulting hypothesis building:

- The case-based method provides the correct most similar case with the correct diagnosis.
- The calculated diagnosis is incorrect.

There is nothing to do with the first line (nothing to learn either!), only the second line is interesting. You then have to adjust the weights of the relevance matrix so that one of the other cases is the most similar. Whether its diagnosis fits is another matter. The process continues until the weights of the matrix remain stable and for all xamples the diagnosis is correct.

Make predictions and get rich

Perhaps you have already asked yourself what expert systems can actually still be used for. If diagnoses are possible, what are the limitations?
The areas of application of expert systems can be found wherever learning systems are used, i.e. where ...

- mathematically precise models are either not known or do not exist
- the complexity of the model makes an exact calculation impossible.
- fast results are required, even if not all boundary conditions are known.
- ...

Yes, I know that some students would reply that all three criteria apply to homework tasks. However, I was referring more to, for example

- weather forecasts
- models about the development of share prices
- statements about the reactions of political decisions to the economy

...
In short, where things get really exciting and we don't know what to do with traditional methods, artificial intelligence can unfold to its heart's content

The areas that behave chaotically are particularly problematic. Where small changes in the initial conditions lead to large changes in the final result, even the most sophisticated expert system cannot help you.
It's like the single grain of sand that triggers a slide: you know that this can happen, but you never know exactly when the pile is big enough...

In this chapter:

- Understanding the basic idea of neural networks
- Building simple networks yourself
- Increasing complexity through feedback
- Understanding important laws of neural networks
- Getting to know applications

Chapter 45 Artful neural networks

In this chapter, you will learn the most important findings in connection with the extremely remarkable field of neural networks. The basics are very easy to understand, but even more difficult questions will not cause you any problems after reading it.

Copying is better than studying

If you think the construction of artificial neural networks has only been tackled in recent years, I have to disappoint you. It started as early as the 40s of the last century, i.e. around the same time as the breakthrough in the construction of the first computers.

The idea is by no means absurd. When I try to bring mental performance to a machine, it makes sense to use the most powerful natural model, our brain, as much as possible.

Of course, this is not quite as easy as it sounds. How many neurons (nerve cells) are present in the human brain is still not one hundred percent clear. However, it will be eighty to one hundred billion, according to current estimates. But you can imagine that no one counted it exactly. The actual computing power of our brain comes about through the connection of neurons with each other, via the so-called synapses and the massive parallelism of processing that this makes possible. There are almost 10^{15} of them - once again only estimated - almost a quadrillion, which can make you quite dizzy. If you assume 10^{11} neurons rounded up, you can expect an average of about 10,000 synapses per neuron.

In individual cases, however, more than 100,000 synapses per cell are also possible. The overall computational capability of our brain arises from the fact that relatively simple nerve cells, each with a manageable computing power, achieve a gigantic performance through their combination. Our brain is, as often emphasized, the most complex organ in the known universe.

Although the brain makes up only a few percent of our body mass, it requires about one-fifth of the basal metabolic rate in energy. So instead of exercising, just take a walk in your mind! But just don't tell your doctor that I kept you from exercising.

Well, that went a bit fast. Take a closer look at a neuron. There are different types of them in our body. To keep it simple, let's simplify it again and assume that a neuron is a summator with numerous inputs (corresponding to the dendritic synapses) and several outputs (through the cell's axon).
The cell has two (binary) states, it can either

- fire or
- be at rest.

Only, when the neuron fires, is this signal transmitted through the axon to all subsequent cells. Bio-electrically speaking, the neuron takes on an action potential when firing, otherwise assumes a resting potential.

But when does it fire and when doesn't it? Now the adder comes into play. Just imagine that the neuron adds the signals of all the firing neurons connected at the input. If a certain threshold is exceeded, it starts firing on its own; otherwise, it remains inactive.

A neuron cannot change the amplitude (strength) of the action potential when firing, but it can change the frequency (rate) at which it is fired. (up to 500-times per second). However, we want to avoid such details in the modeling.

In Figure 45.1, I have shown you how something like this could look schematically.

The inputs x_1 to x_n are - in principle - simply added together. Since we initially only consider binary states of the incoming input neurons, only the values 0 (at rest) and 1 (firing) are possible.

However, the entrances are not equal! The dendritic synapses (symbolized as small clusters at the inputs in Figure 45.1) give weight to the input signal. Each input can have a different weight (w_i), and negative weights are also allowed. This was also "copied" from nature.

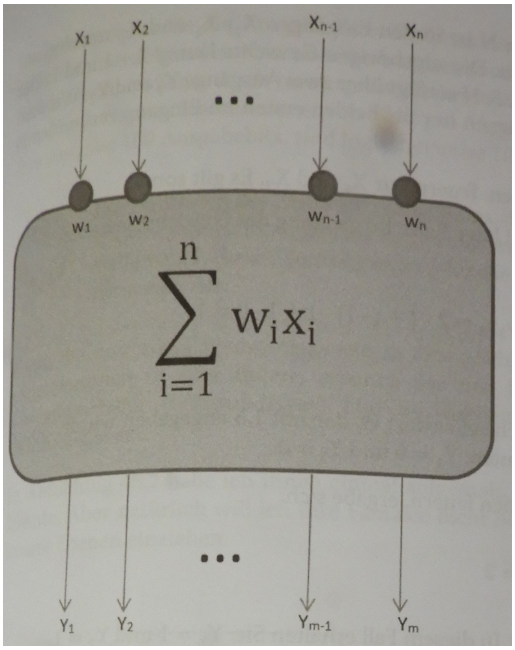


Figure 45.1: Functional Diagram of an Artificial Neuron

In fact, neuroscientists talk about

excitatory and

inhibitory Synapses

With signals through excitatory synapses, firing is stimulated, while with inhibitory ones, it is inhibited. This corresponds to positive and negative weights when summing the inputs.

We don't stress about the exits, though. There, only 0 (idle) or 1 (fire) should be possible (for now), and it should be identical at all outputs Y_1 to Y_n. Whether the neuron fires or not depends on whether the sum of the inputs exceeds a certain threshold, let's call it the Greek letter Θ (Theta):

$\sum_{i=1}^{n} W_i X_i > \Theta$ -> Neuron fires

$\sum_{i=1}^{n} W_i X_i \leq \Theta$ -> Neuron remains at rest

This is quite simple, as the following example demonstrates.

Assuming a neuron N is connected to the inputs X_1, X_2, and X_3 with their respective neurons. The corresponding weights are 2, -1, and 1. The threshold Θ is at 1.5. N has two outputs Y_1 and Y_2. What values do Y_1 and Y_2 have when only the first two connected inputs fire?

As can be inferred from the text, only X_1 and X_2 are firing. It follows that:

$X_1 = 1$, $X_2 = 1$, and $X_3 = 0$. Taking the weights into account, you get the following sum:

$$\sum_{i=1}^{3} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3 = 2 \cdot 1 + (-1) \cdot 1 + 1 \cdot 0 = 1$$

This value (1) does not reach the threshold Θ, which is specified as 1.5. Thus, N remains at rest and the outputs are: $Y_1 = 0$ and $Y_2 = 0$.

If, on the other hand, all input neurons fired, it would result in:

$$\sum_{i=1}^{3} w_i x_i = 2 \cdot 1 + (-1) \cdot 1 + 1 \cdot 1 = 2$$

that exceeds the threshold. In this case, you will get: $Y_1 = 1$ and $Y_2 = 1$. A whole network of neurons forms an artificial neural network. (KNN).

You may consider all neurons in an artificial neural network as interconnected, albeit with different weights. An input weight of 0 corresponds to a missing connection!

Forward to the interconnected networks

If you want to use a KNN for a specific task, you need to feed your input values (consisting of ones and zeros) into the network somewhere. The input neurons are responsible for this, whose inputs are not connected to other neurons. Typically, you will only assign one (binary) input value per input neuron. If you want to input 1,000 bits, your neural network should therefore have over 1,000 input neurons.

There are also biological models for input neurons. In our eyes, there are sensitive photoreceptors on the retina that convert incoming light into electrical signals and transmit these as input to the optic nerve. Our other sense organs also function in a similar way.

Conversely, you read the output of your neural network at the output neurons. These are not connected to subsequent neurons, but instead send their signal out of the ANN, to the "outside." Here too, multiple outputs per output neuron make no sense. You must therefore settle for one value per output neuron. If your task requires 100 output bits, logically 100 output neurons are required.

The neurons of the central nervous system, which ultimately cause muscle contraction, send their signal to the motor end plates of the muscles. Thus, these motoneurons (responsible for motor functions) also serve as output neurons.

If you now bring all the input neurons into one "layer" (level) and the output neurons into another, you will have an (Input-Layer), an (Output-Layer), and hidden layers will form in between. (Hidden-Layer).

In Figure 45.2, I have illustrated how it could look with three layers. But of course, I don't want to limit your imagination; you are welcome to incorporate multiple inner layers as well.

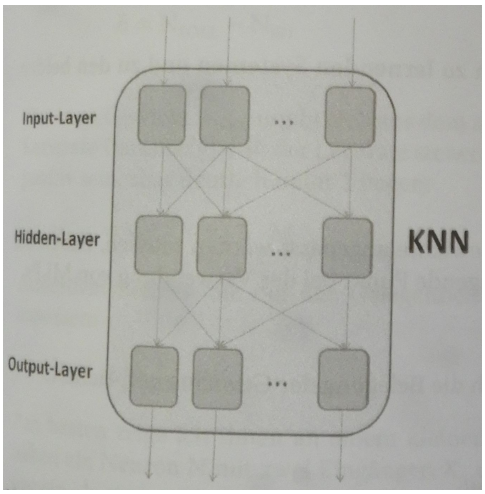


Figure 45.2: A KNN with three layers

Additionally, you are allowed to connect each neuron of a layer with all the neurons in the layer below it. In Figure 45.2, I have only included a few of these connections.

Nevertheless, there is a very important limitation: All arrows always point downwards, which means it is a forward-chained KNN.

Such networks have some advantages when it comes to their implementation. Imagine you lay a pattern (of bits) on the input layer. Then you will immediately know how long it takes before you can read the solution in the output layer (namely, the computation time per neuron multiplied by the number of layers).

Rosenblatt's Theorem

What you see in Figure 45.2 is essentially the construction of a neural network, as it was already developed in 1957 by the American psychologist and pioneer of artificial intelligence, Frank Rosenblatt. He named his baby Perceptron. (from perceive). He started without a hidden layer! His idea was to model the perception of the (human) eye and recognize letters in the process. The input neurons, which you can imagine arranged in a square, were simply fed with the bit pattern of the respective image.

In the training phase, the weights were adjusted so that the output (each specific neuron per letter) matched exactly.

In the classification phase, however, new images of letters were presented, which the perceptron was then supposed to recognize.

Nowadays, we naturally use such perceptrons with many layers. It's called "Multi-layer Perceptron, or MLP for short. MLPs are thus special KNNs where the neurons are arranged in layers and the signal flow only occurs in one direction.

Further general information on learning systems and the two phases can be found in Chapter 43.

Rules for Learning

If you are now wondering how exactly the weights need to be adjusted, you have been paying very close attention. Because that is the crux of the matter when using MLPs.

The knowledge of an MLP is stored through the assignment of weights.

That sounds good, but it has a catch:

MLPs and KNNs in general represent their knowledge sub-symbolically, which means you cannot simply read off what experience your KNN has already gathered.

Chapters 43 and 44 deal with symbolic learning systems, where knowledge is represented in human-readable form.

There are two famous types of learning rules. One corresponds more closely to the biological model and is called the Hebb rule. It was already formulated in 1949 by the Canadian psychologist Donald Hebb.

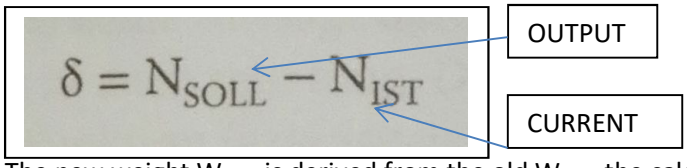
The other is easier to implement and is called the Delta rule. It dates back to the American electrical engineers Bernard Widrow and Marcian Hoff from the year 1960.

The ***Hebb rule*** states that the weight of two connected neurons must be increased when they fire together.

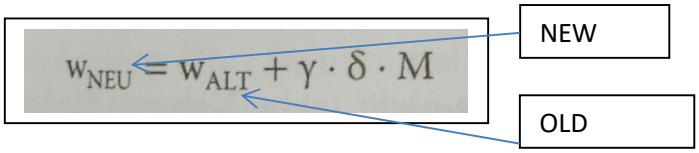
The ***Delta rule*** states that the weight of two connected neurons must be adjusted in such a way that the difference (the "delta") between the existing and the desired output is reduced.

The delta rule also includes a calculation method for its implementation.

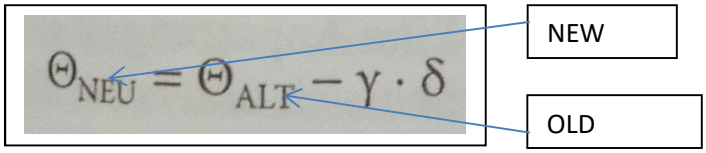
Let M and N be two interconnected neurons, the associated weight between them (that is, at the input of N, which belongs to the output of M), be w. Delta (δ) is calculated from the desired state N_{OUTPUT} of N minus the actual state $N_{CURRENT}$:



The new weight W_{NEW} is derived from the old W_{OLD}, the calculated delta, and a learning rate gamma. (γ). With the learning rate, you control the learning speed. The value should be positive, but significantly below 1:



Additionally, you can also adjust the threshold Theta (Θ) of N on this occasion:



It's best if I show you a small example of how it works. Let's assume you have a neuron N with two inputs X_1 and X_2. They now want your neuron to behave like a logical AND.

As a reminder, I will quickly provide you again with the corresponding truth table (of the conjunction):

Chapter 6 deals extensively with Boolean algebra.

In the following example, you will see how to train the neuron to implement an AND using the Delta rule.

A neuron N is to be trained on the conjunction. The corresponding initial weights, just like the threshold Θ, are all 0. We assume a sample learning rate of γ = 0.2.

The first input pair is $X_1 = X_2 = 0$. The summation in N results in 0 • 0 + 0 - 0 = 0, which does not exceed the threshold (0 = 0). N remains at rest (N = 0). This also corresponds to the OUTPUT value. Since Delta calculates the difference and CURRENT value, it results in: δ = 0. If delta is zero, the new values of the weights and the threshold remain unchanged.

You achieve the same result for $X_1 = 0$, $X_2 = 1$ as well as $X_1 = 1$, $X_2 = 0$. N stays calm each time, which is good.

Only when $X_1 = 1$, $X_2 = 1$ does something new happen. Due to the weights, the sum 1 • 0 + 1 • 0 = 0 remains at zero, which does not exceed the threshold remains at rest, even though it should fire (N_{OUTPUT} = 1, $N_{CURRENT}$ = 0). Thus, it results in;

$\delta = N_{OUTPUT} - N_{CURRENT} = 1\text{-}0 = 1$. From this, the two weights are calculated equally as:

$$w_{NEU} = w_{ALT} + \gamma \cdot \delta \cdot X_1 = 0 + 0.2 \cdot 1 \cdot 1 = 0.2$$

NEW

OLD

The new threshold is:

$$\Theta_{NEU} = \Theta_{ALT} - \gamma \cdot \delta = 0 - 0.2 \cdot 1 = -0.2$$

NEW

OLD

If you now create the same input pair again, you will get the sum in N:

1 • 0.2 + 1 • 0.2 = 0.4, which clearly exceeds the threshold of -0.2 and leads to the firing of N (N = 1).

All good? No way! Because now the other values no longer match. For example, take the pair $X_1 = 0$, $X_2 = 1$. You get when calculating N: 0 • 0.2 + 1 • 0.2 = 0.2, which clearly exceeds the negative threshold and leads to N = 1. The value for this input pair (according to Table 45.1) is, however, 0. An adjustment is you receive as Delta: $\delta = N_{OUTPUT} - N_{CURRENT} = 0 - 1 = -1$. Now, different inputs of the new weights of the two inputs arise:

Eingang 1 : $w_{NEU} = w_{ALT} + \gamma \cdot \delta \cdot X_1 = 0.2 + 0.2 \cdot (-1) \cdot 0 = 0.2$

Eingang 2 : $w_{NEU} = w_{ALT} + \gamma \cdot \delta \cdot X_2 = 0.2 + 0.2 \cdot (-1) \cdot 1 = 0.$

NEW

OLD

INPUT

X_1	X_2	X_1 AND X_2
0	0	0
0	1	0
1	0	0
1	1	1

Table 45.1: Truth Values of the Conjunction (AND)

™The new threshold is:

$$\Theta_{NEU} = \Theta_{ALT} - \gamma \cdot \delta = -0.2 - 0.2 \cdot (-1) = 0.$$

NEW

OLD

In this configuration, however, the input pair (1,0) no longer works: It will incorrectly produce the output 1. Does that mean the task is unsolvable? No way! For example, the weights 0.2 for both input values and the threshold TH= 0.3 would indeed fulfill the task. And there are countless other solutions.

The crucial question is: Will our algorithm ever find a solution?

The answer is "Yes," because the so-called Rosenblatt theorem applies:

Rosenblatt's Theorem

Frank Rosenblatt formally proved that the weights for any function that his perceptron can represent at all stabilize in a finite number of steps. One also speaks of a ***Convergence theorem***.

If you take a closer look at the Rosenblatt theorem, you will find that it actually makes no statement about which functions the network can represent at all. That is also very clever. For if we had chosen the XOR function (exclusive OR) instead of the AND function in the last example, we would have been disappointed.

The XOR problem

A single neuron cannot represent an XOR at all. What is the reason for that? The answer leads you directly to linear separability. Linear separability is the multidimensional, abstract formulation of a very simple geometric fact: Are you able to separate the solution responses using a line? Refer to Figure 45.3 for this. On the left, I have graphically plotted the logical AND, and on the right, the XOR. On the left, a single line (dashed) is enough to separate the zeros from the ones. It will be difficult for you to succeed on the right.

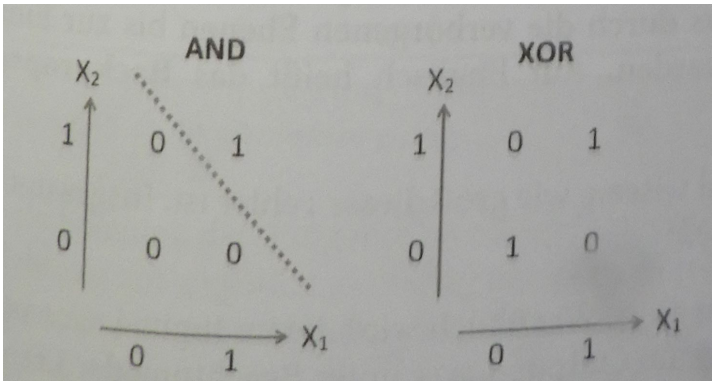


Figure 45.3: AND and XOR - possible and impossible linear separation

Is that bad? Not really. Because using AND and NOT (both of which are linearly separable), you can construct any arbitrary Boolean function.

That AND and NOT form a basis is explained in more detail in Chapter 6.

Historically, however, a resulting controversy led to the KNN research being in a state of dormancy for a long time (from the late 1960s to the early 1980s).

But let's return once again to linear separability. Just imagine a neuron N with the two inputs x and y and the threshold b. The weight at input x is -m ("minus-m"), that at input y is 1. Then the state of the following calculation results:

x • (-m) + y • 1 > b > y > m - x + b

y = m • x + b is the **general equation of a line**! Thus, neuron N separates all values above the line (greater than y) from those below it.

Progress through Backpropagation

XOR is not linearly separable, at least not with a straight line. It's not a big deal if you can't manage with a single neuron, just use two or three or an entire network of them! Each corresponds to a partition wall; with enough of such walls, you can separate all possible discrete point distributions.

But we still have to come to a rather difficult problem now. How do you actually adjust the weights of the inner neurons of a multi-layer network? Here, the error - the discrepancy between OUTPUT and CURRENT - must be propagated, so to speak, "from the bottom," that is, from the output layer backward through the hidden layers to the input layer. In English, that's called Backpropagation.

First, you obviously need to know how big this error is. The weight adjustment is carried out in three steps:

Forward Pass: As usual, the input created in the input layer is passed from top to bottom to the output layer, taking the learning rule into account.

Delta-Investigation: Now simply compare the achieved output (CURRENT) with the desired OUTPUT (we are in the training phase here, where you know the output pattern to be trained for each input pattern).

Backward-Pass: This is the crucial step. The error, that is, the difference between the desired and the actual, is gradually propagated upwards from the output layer to the input layer. Weight adjustments are taking place at all levels.

Phew, that sounds exhausting! But it's even worse: How do you know which weight adjustment in the backward pass is the right one? The neurons are so intricately interconnected that it is not at all clear what effect changing a weight will have on the error value at the very end of the network.

To calculate this, we need advanced mathematics, particularly the gradient method. (siehe Kasten sowie die Mountaneer-Method in Kapitel 42).

Gradient method

With the gradient method, you determine in which direction to search for a relative maximum of a given function.

First, consider a simplified situation where only a single variable (x) is at play. (Abbildung 45.4). With the dashed line, you find the (arbitrarily chosen) starting point, relative to the x-axis.

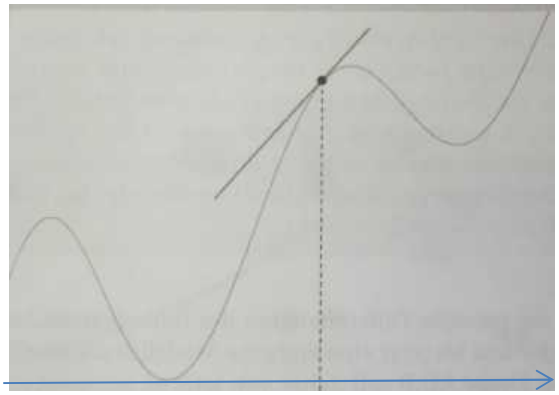


Figure 45.4: Slope of the Tangent 1

Because the slope of the tangent at this point is positive, it is best to look to the right (in the direction of increasing x-values) for a relative maximum of the function curve. If the slope were negative, you should look to the left.

In the case of two variables, things get a bit more complicated.

Again, we start from a random point. The next best relative maximum from the marked point is in the direction of decreasing x-values, but at the same time in the direction of increasing y-values. This corresponds to the respective signs of the partial derivatives of the represented two-dimensional function.

Regarding x, it is a negative value, whereas with respect to y, it is positive.

The vector; which is composed of the partial derivatives with respect to the variables, is called the gradient The gradient overall "points" in the direction of the steepest ascent. If it is an error function (where a minimum is sought), the vector (in all its components) must be negated.

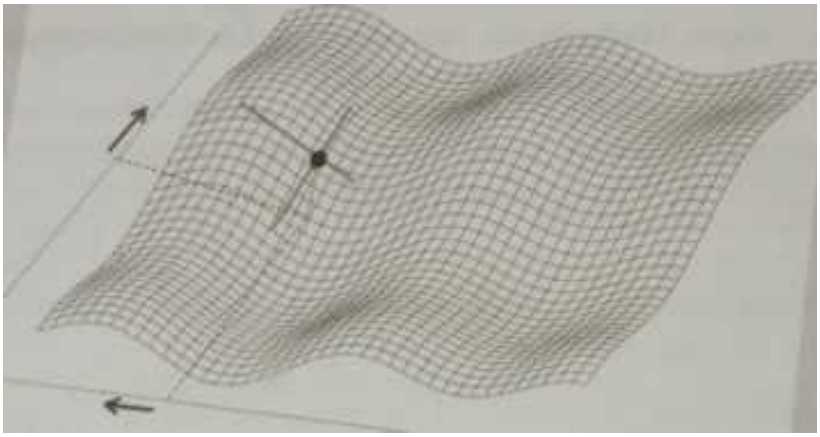


Figure 45.5: Gradient Method with Two Variables

The gradient method (also known as the gradient descent method in the case of minimum search) proceeds step by step. It calculates the partial derivative and changes the current position by a small amount in the determined direction. From there, a new gradient is determined, and the game starts all over again. The procedure ends as soon as a specified maximum number of iterations is reached or the absolute error achieved in the meantime falls below a specified threshold. In doing so, we only very rarely actually reach a Zero point.

The gradient method requires the partial differentiation of the error function. Unfortunately, the use of the threshold, which has allowed us to easily model a neuron until now, is unsuitable for this. Our MLP should therefore tick a little differently from now on. We completely forgo the threshold and instead opt for a differentiable squasher.

Squeeze me!

Still, a neuron is primarily a summator that assigns weights to the inputs and determines a value from them. This part of the artificial neuron N is called the activation function Z:

$$Z_i = \sum_{i=1}^{n} w_i x_i$$

The value of Z_i can be quite varied. Imagine hundreds of weighted input values (negative or positive) need to be added together. The purpose of the squasher S is now to turn Z_i into a small number (between 0 and 1) without using a threshold. Theoretically, the result obtained could be further processed by an output function, but for simplicity, $S(Z_i)$ is already to be the output of neuron i.

There are several ways to get a differentiable squasher. The logistic function is often used. It has the following principal representation:

$S(Z)=1 / (1 + e^{-z})$

As you can see in Figure 45.6, the real (very small to very large) values are squeezed into the range between 0 and 1.

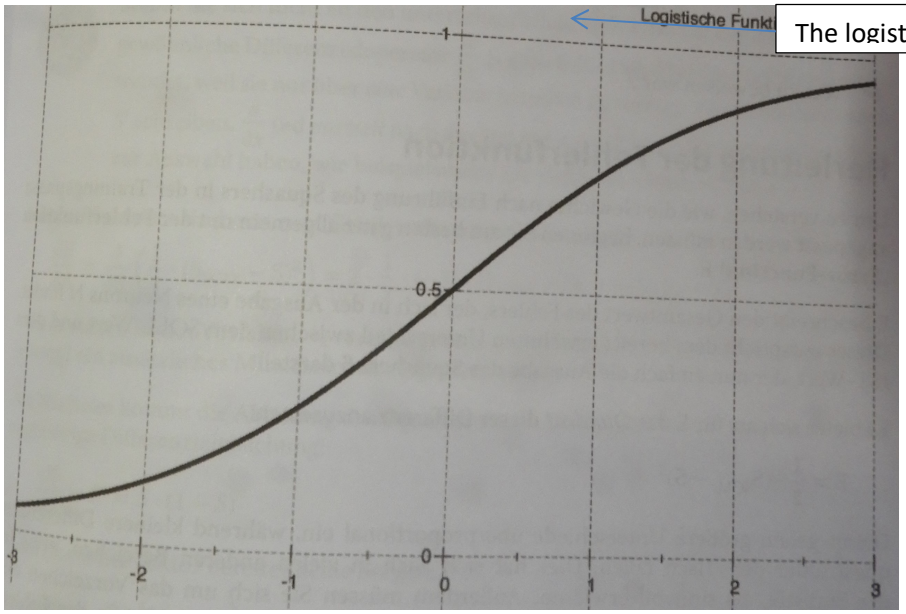


Figure 45.6: The logistic function as a squasher

For Z = 0, S(Z) = 1/2 Another highly desirable property of the logistic function arises from the following differential equation:

$S(Z) = S(Z) - (1-S(Z))$

You can thus easily calculate the value of the derivative of S(Z) using a subtraction and a multiplication from the already existing value of S(Z). Very practical!

If you are interested in the proof:

It applies under the use of the quotient rule of differentiation:

$$S'(Z) = \left(\frac{1}{1+e^{-Z}}\right)' = \frac{0-(-1)e^{-Z}}{(1+e^{-Z})^2} = \frac{e^{-Z}}{(1+e^{-Z})^2}$$

You will receive the same for:

$$\begin{aligned} S(Z)\cdot(1-S(Z)) &= \frac{1}{1+e^{-Z}}\cdot\left(1-\frac{1}{1+e^{-Z}}\right) \\ &= \frac{1}{1+e^{-Z}} - \frac{1}{(1+e^{-Z})^2} \\ &= \frac{1+e^{-Z}-1}{(1+e^{-Z})^2} \\ &= \frac{e^{-Z}}{(1+e^{-Z})^2} \end{aligned}$$

What was to be proven...

Derivation of the Error Function

To understand how the weights need to be adjusted after the introduction of the squasher during the training phase, it is best to start quite generally with the error function E. E describes the total value of the error found in the output of a neuron N. This corresponds to the aforementioned difference between the OUTPUT value and the CURRENT value, which now simply represents the output of the squasher S.

It is advisable to set the square of this difference for E:

$$E = \frac{1}{2}\cdot(S_{SOLL} - S)^2$$

OUTPUT

This means that larger differences are disproportionately included, while smaller ones are almost overlooked. This has also proven to be useful in many other areas, such as statistics. Additionally, you no longer need to worry about the sign of the difference (ultimately, for the magnitude of the error, it doesn't matter whether the output is too large or too small). I only slipped the factor of 1/2 in there so that it could be canceled out by the "2" when differentiating. But that's not a problem. Make it clear using Figure 45.5 that any (but positive!) prefactor is not relevant when searching for the direction of the gradient. The mountains and valleys may become higher or flatter, but they remain in the same place. Finally, the gradient method comes into play. The question is: change the input weights w_1 to w_N of a neuron N to at least slightly reduce E? Again, the learning rate Gamma (γ) comes into play. Additionally, the negative sign is supposed to ensure that we are concerned with minimizing the error (otherwise, we would be determining a maximum, not very clever...).

For each weight w_i from N, you set:

$$\Delta w_i = -\gamma \cdot \frac{\partial E}{\partial w_i}$$

Unfortunately, E is not directly dependent on w_i, but indirectly through the squasher S. This one, in turn, operates on the activity function Z.

Since E refers to the sequential execution of Z and S, you need the chain rule of differentiation:

$$\Delta w_i = -\gamma \cdot \frac{\partial E}{\partial w_i} = -\gamma \cdot \frac{\partial E}{\partial S} \cdot \frac{dS}{dZ} \cdot \frac{\partial Z}{\partial w_i}$$

Don't be bothered by the different symbols in the derivation. The ordinary differential operator $\frac{d}{dx}$ (»d by dx«) is applied to the function S because it only has one variable Z). You are also allowed to write S' for that. $\frac{\partial}{\partial x}$ (»d partial with respect to dx«) is necessary when you have multiple variables to choose from, such as the many weights w_i of Z ...

All three factors are now fortunately determined quite quickly:

$$\frac{\partial E}{\partial S} = \frac{\partial}{\partial S}\left(\frac{1}{2} \cdot (S_{SOLL} - S)^2\right) = 2 \cdot \frac{1}{2} \cdot (-1) \cdot (S_{SOLL} - S) = -(S_{SOLL} - S)$$

As expected, the prefactor 1/2 has disappeared. For this, an additional minus sign has crept in (due to the internal derivation).

Next up is the derivation of the squashers to Z. For that, we use the corresponding differential equation!

$$\frac{dS}{dZ} = S' = S \cdot (1 - S)$$

The third factor also poses no concerns:

$$\frac{\partial Z}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\sum_{i=1}^{n} w_i x_i\right) = x_i$$

From the large sum, only the summand $w_i x_i$ depends on w_i, all others are derived to zero. And also from $w_i x_i$, after the partial derivative with respect to w_i, only x_i remains.Overall, you will thus receive:

$$\Delta w_i = -\gamma \cdot \frac{\partial E}{\partial S} \cdot \frac{dS}{dZ} \cdot \frac{\partial Z}{\partial w_i} = \gamma \cdot (S_{SOLL} - S) \cdot S \cdot (1 - S) \cdot x_i$$

Weight adjustment of a neuron in the output layer.

For an output neuron N, the situation is simple. You do indicate yourself during the training phase - S_{OUTPUT}. This allows you to immediately determine the error signal:

The expression $\delta_N = (S_{OUTPUT} - S) \bullet S \bullet (1 - S)$ is called the error signal of the output neuron N.

The formula for adjusting the weights w_i of N is as follows:

$$w_i^{NEU} = w_i^{ALT} + \Delta w_i = w_i^{ALT} + \gamma \cdot \delta_N \cdot x_i$$

Note that Δw_i can also be negative. Thus, a weight adjustment upwards and downwards is possible.

Weight adjustment of an internal neuron

The situation looks more interesting with an internal neuron N. Where do you get the value of S_{OUTPUT} from? You determine this from the error signals of the connected subsequent neuron!

The expression $\delta_N = F \cdot S \cdot (1 - S)$ is called the error signal of the inner neuron N.

The factor F is determined from the weighted sum of all error signals of the subsequent neurons M:

$$F = \sum_M \delta_M \cdot w_M$$

w_M refers here to the input weight of M that is connected to N. The formula for adjusting the weights wi of N is as follows:

$$w_i^{NEU} = w_i^{ALT} + \Delta w_i = w_i^{ALT} + \gamma \cdot \delta_N \cdot x_i$$

Oh, that sounds very confusing. It's best if I show you this with a specific situation, then the formulas should hopefully become a bit clearer.

Consider Figure 45.7 in this regard.

Your task is to adjust the weights w_A, w_B, and wc of the inner neuron N. Through the forward pass, you know the corresponding input signals that depend on the respective output of neurons A, B, and C. That's clear so far.

Because N is not an output neuron, you need the error signals from the neurons X, Y, and Z connected behind it for weight adjustment. These, in turn, require their subsequent signals. It keeps going like that, all the way to the output layer. You can calculate their error signals immediately.

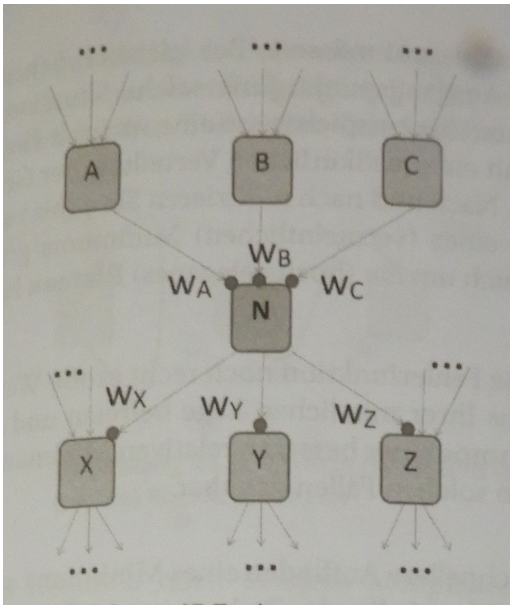


Figure 45.7: Initial situation in an internal neuron N

Therefore, it is necessary to proceed from bottom to top in the backward pass.

Assuming you have already determined the error signals (δ_x, δ_y, δ_z) from X, Y, Z. Then calculate the error signal S_N from N as follows.

First, you sum the weighted successor error signals and obtain F.

$$F = w_X \cdot \delta_X + w_Y \cdot \delta_Y + w_Z \cdot \delta_Z$$

This results in the error signal of N:

$$F = w_X \cdot \delta_X + w_Y \cdot \delta_Y + w_Z \cdot \delta_Z$$

With that, you can calculate the adjustments of the weights of N:

$$w_A^{NEU} = w_A^{ALT} + \Delta w_A = w_A^{ALT} + \gamma \cdot \delta_N \cdot x_A$$

$$w_B^{NEU} = w_B^{ALT} + \Delta w_B = w_B^{ALT} + \gamma \cdot \delta_N \cdot x_B$$

$$w_C^{NEU} = w_C^{ALT} + \Delta w_C = w_C^{ALT} + \gamma \cdot \delta_N \cdot x_C$$

Various Variants

The backpropagation methods often work very well in the presented variant, but occasionally difficulties do arise.

Refer to Figure 45.4 again. What happens if your step (in the right direction) is a bit too big? Then you will land on the other side of the "summit" and would march back again

With the same stride length, you might even return to the starting point. Such a situation is called oscillation. To avoid them, you could, for example, use a variable learning rate γ. You start your algorithm with a random distribution of weights and a fairly large learning rate, about γ = 1/2. Gradually reduce γ (down to a value of γ = 0.01) as you approach a (presumed) minimum. Of course, it could actually be a (high-altitude) plateau. There is a solution for that too.

If your algorithm terminates, even though the error function still shows relatively large values, a spontaneous "jump" could free you from your predicament and lead you to another point from which you can move towards a better relative minimum. An increase in the learning rate would also be conceivable in such cases.

A larger learning rate allows for faster finding of a minimum and helps overcome stagnations in learning. If the goal is "further away," a larger learning rate is recommended.

A lower learning rate reduces the risk of oscillations and is especially appropriate when dealing with a highly fluctuating error function. To find a minimum "nearby," the learning rate must not be too large.

Another option is to add a stabilizing term, also known as momentum (analogous to the physical moment of inertia).

This momentum should ensure that a previous weight adjustment also affects the next one. For example, if an adjustment with Δw = -0.2 was made in the last step, a (small) part of this reduction (about -0.02) should be included again in the calculation in the next step. This ensures certain continuity is maintained and a zigzag course in the gradient method is avoided.

The Power of Feedback Loops

Forward-connected neural networks like MLPs are quite amusing, but do you have any idea of the immense possibilities that arise from feedback loops?

The fully connected network already described, where there is a connection from every node to every other node, is also a special case of an RNN with feedback, which is also referred to as a recurrent RNN.

An artificial neural network with feedback is called recurrent.

To keep things somewhat organized, we should first distinguish some basic types of feedback.

Figure 45.8, for example, shows you (in bold) on the left side a direct feedback loop, where the output of a neuron is also another input. On the right, you see a lateral feedback where there is a cross connection within the same level.

Figure 45.8, for example, shows you (in bold) on the left side a direct feedback loop, where the output of a neuron is also another input. On the right, you see a lateral feedback where there is a cross-connection within the same plane.

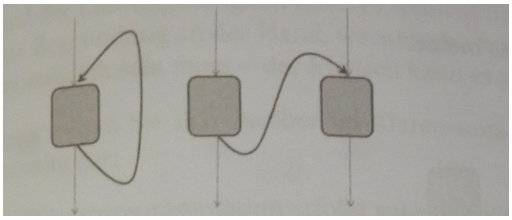


Figure 45.8: Direct (left) and lateral (right) feedback

Of course, it can get much more complicated. An indirect feedback loop represents a branching back to a previous level. (Figure 45.9)

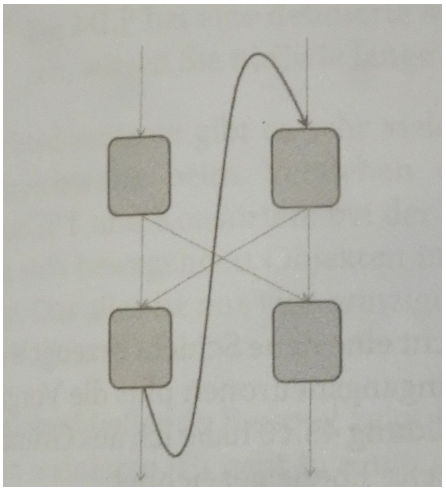


Figure 45.9: Indirect Feedback

To give you a little insight into what you can actually achieve with feedback, I will first present a problem that you will not be able to solve with pure MLPs.

Neural networks are sometimes used to predict stock prices. So, you feed your MLP every day with the current value of stock A. On Monday, the price is €100, on Tuesday €90, on Wednesday €80, on Thursday €90, and on Friday €95. The value of the paper is the same on Tuesday and Thursday, but there is an incredibly important difference for stock traders: On Tuesday, the paper is in a downward trend, while on Thursday, a friendly price development is noted!

How is a purely feedforward KNN supposed to recognize the difference between the Tuesday and Thursday course? On both days, the current input values are identical. It can't remember the values from the previous day.

What you need is a concept that allows you to retain access to "previous" input patterns. How about if you just create a copy of the input neurons? For this you need lateral feedback. Typically, the weights of this layer called the context layer remain constant at 1.

Figure 45.10 shows you what I mean.

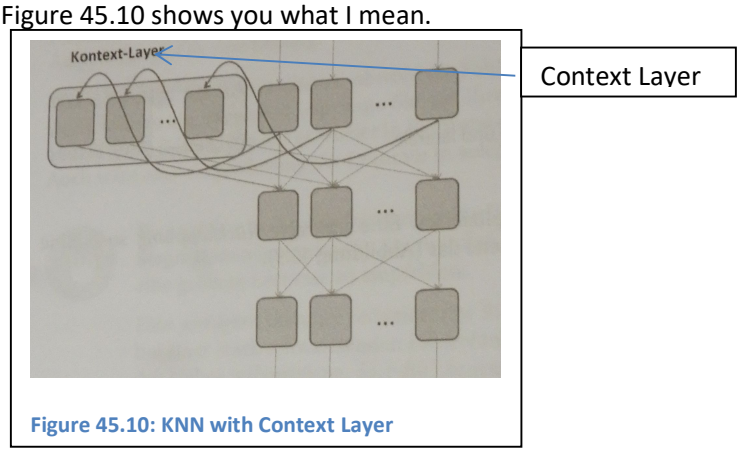


Figure 45.10: KNN with Context Layer

On the left, you can see how a new layer is created from the outputs of the input layer. The second layer now has full access to all input neurons plus their predecessor, namely the context layer. In Figure 45.10, I have only drawn a few exemplary arrows to the second level for clarity.

The input weight of the thickly drawn arrows of the lateral feedbacks remains unchanged at 1 in this case. Only in this way can it be ensured that it is truly the exact copy of the input neurons from the predecessor pattern.

With this method, you essentially travel back in time and solve problems that require context. You can imagine that you can generate additional context layers with even older input patterns in the same way. In the meantime, numerous variants of recurrent networks have been invented to realize different types of memory.

Limitless fields of application

Here and there, I have already hinted at what this whole effort is useful for. In this final section, I would like to present a few examples that should inspire you to design your own networks to realize your ideas.

Generally, it can be said that KNNs are always used where...

- a symbolic realization would not be efficient,
- the complexity of the problem is too great or
- no alternative solutions exist.

Due to the capabilities of our brain, proponents of KNNs always have a good argument for why a highly efficient solution to a specific problem must be possible - after all, humans can do it too!

However, you should be aware that the use of KNNs also has a downside:

- The knowledge representation is subsymbolic, which means you never know if your system has learned enough or not.
- The control of the KNN is very problematic. You never know exactly how your system will react. Therefore, for example, you will not let a nuclear power plant be controlled autonomously by KNN networks.
- An MLP has a defined maximum runtime. If you use a recurrent system, you never know how much longer you have to wait.

Nevertheless, there are many exciting areas where KNNs are used. For example, in understanding or generating natural language, think of ***ChatGPT*** and its counterparts, in recognizing people in images or tracking moving objects in videos, in climate research and weather forecasting: All of this is just a tiny selection of the many fascinating application areas for KNNs.

The final example shows you how easy it is nowadays to train MLPs on data with Python. This is about the identification of iris plants based on the lengths and widths of sepals and petals.

Chapter 31 shows you how to work with libraries in Python. Towards the end, it describes exactly what the Iris dataset is about and how you can view it.

First, let's grab the iris data:

```
from sklearn.datasets import load_iris
iris_db = load_iris()
X = iris_db.data
Y = iris_db.target
```

X now contains the datasets, y the corresponding labels. In our case, these are 3 different types of iris plants.

The structure of the MLP is remarkably simple:

```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(40,35), max_iter=1000)
```

The first parameter controls the architecture of your network. Only the hidden layers are specified. For demonstration purposes, you have created 40 neurons below the input layer here, which are connected to the 35 neurons in the subsequent layer. This, in turn, sends the signals to the output layer.

With `max_iter`, you specify the maximum number of iterations for which your network will be trained, even if the error is still too large at the end (you can also adjust all of this).

```
mlp.fit(X, y)
```

This is the crucial line: This is how your model (the MLP) is fitted to the data! You are probably wondering if there is any missing information about the input or output layer: no! Due to the number of features in X (in our case 4) and the number of classes in Y (here: 3), the extent of these layers is automatically determined.

So that you can see how well the algorithm works, use this code to plot the error curve (loss, the loss value) for each respective iteration.

```
import matplotlib.pyplot as plt
plt.plot(mlp.loss_curve_)
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()
```

The final result can be found in Figure 45.11.

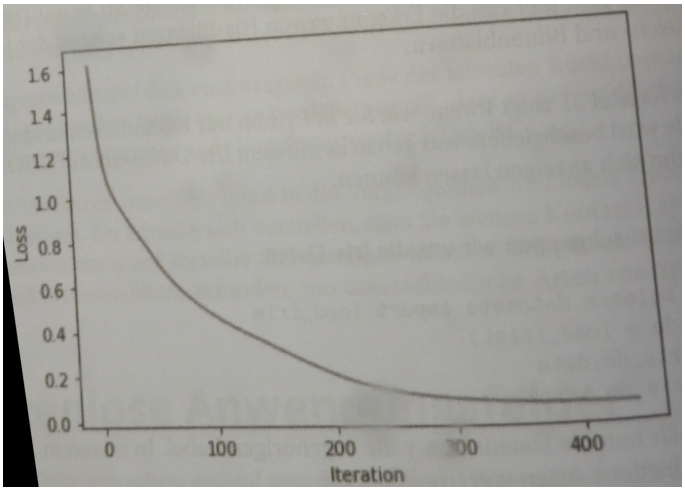


Figure 45.11 Error curve during training

Chapter 47 Designing and creating on the web

Here you will learn how the Internet works and how a series of bits and bytes are turned into colorful, perhaps flashing text and images. After a brief look back at the history of the World Wide Web, it's time to get started. I will explain the HTTP protocol and the HTML and XML languages that everything revolves around.

Web technology for insiders

Until the 90s of the last millennium, the Internet led a dismal existence. Apart from the military, who had driven the network forward, only universities and research institutions were able to benefit from electronic data transmission.

At CERN, Tim Berners-Lee developed a concept to make research results electronically available to colleagues within his institution.

CERN is the abbreviation for “Conseil Européen pour la Recherche Nucléaire”. This multinational research institution is based in Geneva. CERN has gigantic particle accelerators to research ever smaller components of atomic nuclei and the origin of the world.

His idea was to prepare pure text graphically in such a way that images and links to other texts would also be possible. Naturally, he packed his “Hypertext Transfer Protocol” (HTTP) into the already existing protocol architecture of TCP/IP.

Details on TCP/IP can be found in chapter 46.

Within HTTP, the “Hypertext Markup Language” (HTML) is the actual carrier of the information. The payload of HTTP therefore consists of HTML files.

After some back and forth, Tim Berners-Lee decided to name the overall concept the World Wide Web. This is the reason why you still have to enter “WWW” in the browser for many target pages today. However, the server may also be called something else. The important thing is that it can handle HTTP

HTTP in short form

HTTP is the protocol used by the client and server to communicate in order to transfer HTML. The client begins and makes its request. This request is initiated by a program, which you know as a browser, so that you receive the result immediately in a graphical format.

In the browser, you use the URL to enter which page on which server you want to view

URL is the abbreviation for Uniform Resource Locator, i.e. a standardized way of identifying a target resource. Details can be found, as usual, in an RFC (Request for Comments). In this case it is RFC 1738.

The URL consists of several components. It starts with the protocol name (terminated with the character string “: //”). This is followed by the specification of the target computer using the qualified domain name, each divided by dots. This is terminated with a slash “/”. Everything else that follows is used for the “localization of the resource” on the system, i.e. the actual task of the URL (Figure 47.1).

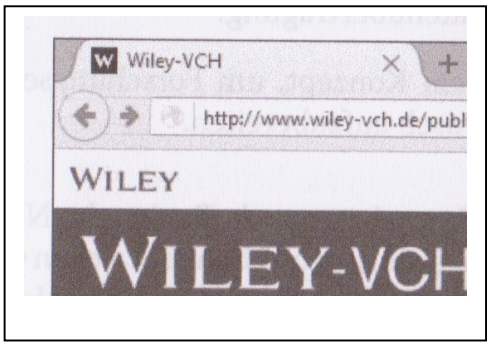


Figure 47.1: URL in a browser

In the past, there were numerous other variants besides http, but these have now become almost meaningless. Even if you simply omit the protocol name, your browser will automatically select the Hypertext Transfer Protocol.

This recognizes various commands. The most important are

- GET: This is used to request a page from the server. The transferred data is part of the URL, so you can recognize it in the link. The maximum length of a URL is 2048 bytes, which also limits GET.
- P0ST: In contrast to GET, this time the data is transferred in the message body, so it is not visible in the link. In addition, any amount of data can be transferred using this method.
- HEAD: Like GET, but only downloads the header of the target page, not the actual data.
- PUT: This is used to upload data to a web server.
- DELETE: This command is used to delete pages on the target system.

You can probably imagine that the last two commands are switched off almost everywhere these days. The web administrator would certainly be happy if every client could upload any data or even delete existing pages.

To give you a feel for how the whole thing works, I'll show you what an HTTP request looks like and what the server responds with a concrete example.

You first request the target system via TCP/IP. If you use the telnet command, enter the port address, in our case normally “80”, separated by a space at the end (Figure 47.2).

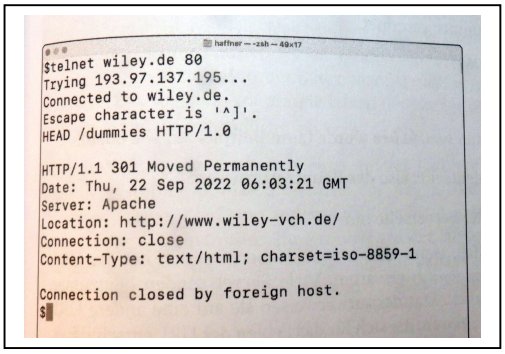


Figure 47.2: Example of an http session

Details on telnet can be found in Chapter 49!

The next three lines (`Trying` ..., `Connected` .. and `Escape` ...) indicate that you have established a connection.

Only in the following line is the the first HTTP command is used: "`HEAD /dummies HTTP/1.0.` This signals to the target system that you want to download the resource /dummies with the proto version 1.0.

Please note that you must type a "double return" after entering the HTTP command!

After the blank line, you will see what the server responds. It first reminds you that the target server is speaking with the protocol version HTTP1.1, but this is not a problem. Conversely, it would be more problematic: the requested protocol should not be higher than the existing one.

This is followed by a status message with the code 301, the meaning of which also appears in plain text: Moved Permanently, which means that the requested page has been moved permanently from this location. Incidentally, the web server is an Apache, `text/html` is accepted as the language, in ISO character encoding 8859-1. This is an ASCII extension.

HTTP status codes

The structure of the codes that you can expect as a response from the server is ordered according to the first digit of the three-digit number.

- 1... stands for pure information.
- 2... signals a success.
- 3... indicates that information has been moved (for example 301 as above).
- 4... is an error on the client side, i.e. the user browser.
- 5... indicates an error on the server side.

The answer 200 is the normal case if everything works. If you have mistyped the URL, you can expect a 404 (*Not found*). However, many servers nowadays no longer send back a 404. Instead, they refer to another URL or display a nicely prepared page that apologizes for the missing URL. The reason is simple: presumably the cause is not that the user made a typing error, but that a link points to exactly this location that does not (or no longer) exist...

But where was the page moved to? The location tells us. However, we initially only requested the HEAD. If you start the same call, but now select GET instead of HEAD, you will receive the body as well as the header - in HTML.

HTML in short form

In Figure 47.3 I show you what this answer looks like. The beginning is exactly the same, so I'll leave it out and concentrate on what's really new

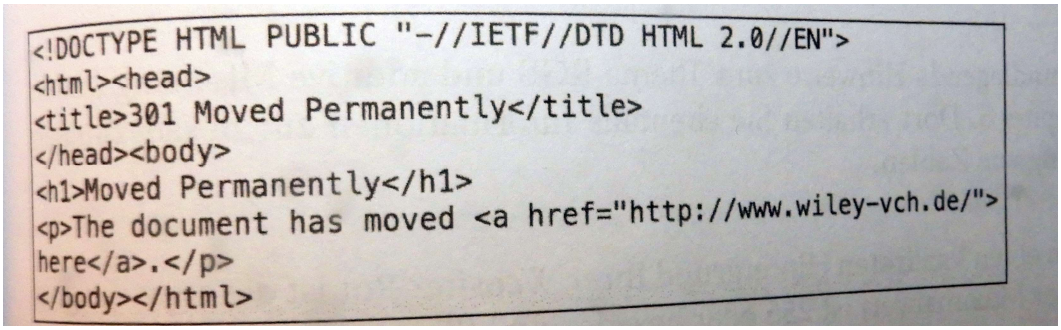


Figure 47.3: Body of an http response

HTML files always start with the initial html tag and end with the same, supplemented by an initial slash "/": `</html>`. The data contained therein is also structured in the same way:

`<head> …..</head>`

`<body> …..</body>`

Even within these subsections, you will again find

`<title> …..</title>, <h1>...</h1>, <p>...</p>` und `<a>...</a>.`

The title is used by the browser in such a way that it is actually the window heading. The h1 stands for a heading of size 1. You can also use other numbers up to 6, but the font size will always be smaller. The browser alone decides how large it is ultimately displayed on your screen. Nowadays, common programs allow the user to determine how large the standard should be.

The p stands for paragraph, which describes a text-only area (not a heading). The tag a deserves special attention. The following `href=` (hypertext reference) refers to another URL, which is displayed as a link, for example. Modern browsers replace the 301 redirect status with the content of the redirected page.

All details on HTTP can be found in RCF1945 (HTTP1.0) and its successors. HTML is described in RFC 1866 (version 2.0) and successor articles.

HTML to XML

So far, the text delivered by the server is anything but colorful. Of course, the browser is ultimately responsible for bringing some color into play. How, in turn, is regulated by the HTML color codes.

The pivotal point is the RGB color display. Here you represent the red, green and blue values of your color as an additive number.

To keep this reasonably clear, one byte is used for each color value, but the specification is hexadecimal.

Some basic information on RGB and additive mixtures can be found in chapter 5, where you will also find information on the hexadecimal representation of numbers.

Do you want a bright red background for your website? Red is the first <\ color, the maximum value is 255 or hexadecimal FF.

```
<body style="background: #FF0000" >
```

Do you want a bright red background for your website? Red is the first <\ colors, the maximum value is 255 or hexadecimal FF.

```
<body style=“background: #FF0000” >
```

Do you want your link to be pure green? It looks like this:

```
<a style=“color :#00FF00”>
```

The font tag can also be used to display colors as text in HTML:

```
<font color=#0000FF>This is blue text. </font>
<font color="blue” >This too.</font>
```

You can also determine the character sets, font size and other properties of your typeface.

Any color mixture is of course also possible. For example, “A30C72” corresponds to a kind of purple, in which the red component A3 is around 63 percent, the green 0C only around 4 percent and the blue component around 44 percent.

The maximum number “`FFFFFF`” results in white, while “`000000`” stands for black,

Cascaded styling

Quite early on in the development of HTML, it was noticed that it is all too tedious to explicitly assign specific properties such as size, color or style to each element of a page. To remedy this, Cascading Style Sheets (CSS) were invented. The idea is to separate the content of a text from its specific appearance. The style sheets contain the adjustable properties in a separate data area, so to speak, while you only refer to the CSS element in the text. This allows you to give all your (perhaps a hundred?) pages the same look with one CSS file.

For example, if you want to give all your headings with the tag h1 the font size 18, the color “red” and the font “Arial”, you can create a file my_styles. css file with the following content:

```
<style type-“text/css”>
```

```
<!--
h1 (
       font size: 18 pt;
       color :#FF0000
       font-family:arial;
(-->
```

In your HTML files, use the link tag to refer to the CSS (Cascading Style Sheets) file:

```
<link rel= “stylesheet” href=“my_styles.css”>
```

Please note that the attribute rel = “stylesheet” is mandatory. You are of course free to choose the file name. However, the suffix css has become generally accepted.

Unlimited possibilities

It's fantastic what you can do with HTML. Of course you can also include images or add a link to them (when you click on them):

```
href= “http: //my, targetaddress.com”>
```

```
<img src=“meinpicture. jpg” alt=“My picture”/x/a>
```

The outer frame is formed by the reference tag a. You already know the meaning of href=. However, an image (image, `img`) is now placed in the position to be clicked. The alternative tag (`alt=`) is only used if, for example, images are switched off on the browser page. Then the text “My image” is displayed instead.

It goes on like this forever. The possibilities of HTML have been extended with every version. There is no end in sight.

Be careful when using non-standard elements. For example, certain browser types allow the `<blink>` tag, which makes a text blink. However, this leads to an error message for others. As a server operator, you do not want to maintain a separate page for each browser version, so you should only use standards.

The specification of the current HTML version (5) can be found at `http://www. w3. org/TR/html5.`

If you take just a small step away from HTML, you will find that you can structure any text very finely using the concept of nested tags. This is called XML - and it doesn't just work on the web.

XML is the abbreviation for eXtensible Markup Language and allows the use of self-defined tags.

Would you like to send messages? How about:

```
<?xml version=“1.0” encoding=“iso-8859-l”?>
<message>
<from>Alice</from>
<to>Bob</to>
<heading>Congratulations</heading>
<text>Happy birthday!</text>
</message>
```

Of course, you can - as before - assign any cascaded style sheets (CSS) to the individual elements to define textual properties

In principle, you will find XML files everywhere. Browsers also display XML files, and without a corresponding CSS, this looks like a collapsible tree (in the individual levels) (Figure 47.4).

No style information appears to be linked to this XML file. The tree view of the document is shown below.

```
- <message>
      <from>Alicc</from>
      <to>Bob</to>
      <heading>Congratulations</heading>
      <text>Happy birthday!</text>
</message>
```

Figure 47.4: Browser view of an XML file without CSS

If you click on the minus sign with the mouse, the tree collapses at the respective level (in this case completely) as shown in Figure 47.5.

```
+ <message></message>
```

Figure 47.5: Collapsed version of the XML file

Of course, I would like to conclude this chapter by showing you how to assign a CSS file to an XML file. To do this, add your CSS file as the second line:

```
< ?xml-stylesheet type=“text/css” href="messages . css” ?>
```

The reference file can be any URL. Its content, in turn, must tell the browser how the individual components should be displayed. In our case, these are the tags:

- message
- from
- to
- heading
- text

For example, the following content of the file message. css file would be conceivable:

```
message {
	background-color: #ABCDEF;
	width: 100%;
}
from, to {
	display: block;
	color: #00FF00;
	font-size: 14pt;
	text-transform: uppercase;
}
heading {
	display: block;
	color: #FF0000;
	font-size: 20pt;
}
text {
	display: block;
	color: #000000;
	margin-left: 10pt;
}
```

Try out the various options for yourself. You already know the colors. The width refers to the current browser window. In the case of block, the display option displays the associated text as if it were in a p environment. You can change the text using commands such as text-trans form. For example, make all letters uppercase. And that's just the tip of the iceberg, try it yourself!

You can find a reference to the various CSS standards here:

```
http://www. w3. org/Style/CSS
```

The result will look like Figure 47.6. Do you like it? You are welcome to try out your own ideas. After all, there's no accounting for taste ...

Instead of taking the somewhat cumbersome route via CSS to display your XML files appropriately, you should also consider the possibilities of scripting languages.

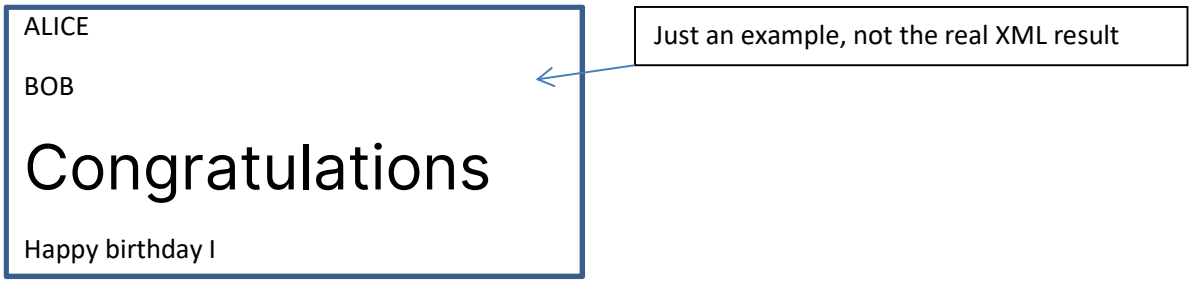


Figure 47.6: XML file with CSS

An explanation of the most important scripting languages can be found in Chapter 48.

Deep Web, Darknet...

One of the basic ideas behind the World Wide Web is to ensure the broadest possible exchange of data across the globe.

Due to the wealth of information available, search engines play a crucial role on the WWW. Companies invest a lot in order to land at the top of the hit list for certain keywords.

However, specialist databases or pages that can only be accessed by entering a form cannot be indexed using conventional web crawlers, script programs provided by search engine operators. Intranets that are only accessible to certain user groups also belong in this category.

This data forms the deep web. It is estimated that the amount of information in the deep web exceeds that in the visible web many times over

The dark web is to be distinguished from this. This refers to the network of computers that do not want to be found by the general public. Technically, these are overlay networks that use existing infrastructure to create their own network topology. In this respect, the darknet also consists of different networks. New members of the network usually have to be invited beforehand. Communication is generally encrypted. As the name suggests, the darknet is a playground for criminal activities and a place for shady business. However, the darknet is also used legally. Think of journalists or whistleblowers whose anonymity must be preserved. The desire to communicate with others - free from surveillance by secret services, not threatened by censorship - justifies the existence of the darknet for many people.

Chapter 48 Scripting languages

This chapter deals exclusively with scripting languages, which have become increasingly important. After a brief introduction, we will take a look at AWK, Perl, PHP and JavaScript. Of course, Python will also be included. This is not a comprehensive course, which would require an entire book for each language. But you should gain an impression of what is possible and how scripting languages are used. That's quite a lot for such a small chapter, so let's not waste any time!

Scripted shell scripts

You can imagine the origin of scripting languages like this:

The administrator of a mainframe computer (which was the term used in the last century - today, every smartphone has a larger computing capacity) was annoyed by having to manually type a number of very similar commands one after the other into the console (i.e. the interface with which he communicated with the computer). “This is a computer,” he asked himself, ”why can't I automate this?”

Of course, writing a program would have been an option, but the endless compiling, debugging, testing and so on takes far too long. “Until then, I entered it by hand much faster ...”. On the other hand - it couldn't go on like this. It should be possible for the administrator (today we would say: every user) to execute small programs directly in the console (all text-based, mind you, windows and mice were not yet available) that do not have to be compiled separately. Nothing big, of course, just a few useful commands such as conditional statements, loops, wildcards (“*”) and stuff like that. In the end, the communication platform (shell) itself should be able to execute such instructions, similar to following a script (screenplay) in a movie. No sooner said than done.

Shell script programming was invented. Since then, all computer consoles have had more or less extensive programming options.

The first real shell can be traced back to the Brit Stephen Bourne, who developed it for the UNIX operating system at the end of the 1970s. This program was automatically started using `/bin/sh` (pronounced “bin, shell”) every time you logged in. This shell was named Bourne shell in his honor.

Later, there were a number of other shells that could be used to do similar things, but with a different syntax. Shortly afterwards, the American Wilton Joy introduced a variant of the shell based on the “C” programming language (`/bin/csh`).

Despite the euphonious name (“C shell” sounds like “sea shell” and means “sea shell”), there were other shell variants. David Korn, for example, introduced the Korn shell (`/bin/ksh`) at the end of the 1980s. It is backwards compatible with the Bourne shell and integrates the most important new features of the C shell.

The next important step was the development of the Bash (“Bourne again Shell”) `/bin/bash`. It contains all the important elements of the others and is now considered the standard for UNIX-like operating systems (including Linux). macOS now has the Z shell as standard, which in turn is compatible with Bash.

Under ***/etc/shells*** you will find a list of all supported shells on the respective system!

However, shell scripts are not only good for processing console commands in a sequential and structured manner. They unfold their strength precisely where you would otherwise have considerable programming effort, especially when it comes to ...

- Manipulating file names and contents,
- calling up other programs or
- time-controlled starting of tasks

Okay, you probably want to see a shell script like this. How about a bash script that creates ten new, empty files, each named File1.txt to File10.txt? Nothing could be simpler:

```
#! /bin/bash
for i in {i ..10}
   do
        touch File$i.txt
done
```

The first line indicates that the content of this file is to be processed with /bin/bash. The for-loop works in such a way that the variable i is assigned the values 1 to 10 one after the other. The do and done commands frame the body of the loop.

The touch command normally updates the access date of a file. However, if this file does not yet exist, an empty file is simply created. As you can see, not many brackets or quotation marks are required as parameters. The $i outputs the "value of i (i.e. the numbers 1 to 10 in sequence). Done'.

In order to run a shell script, you must assign execution rights to the respective file (which should ideally have the suffix sh), which is typically done using chmod +x filename . sh.

You start the script in the current directory with . /filename. sh.

Ten new files are then created. That's all done.

Now you want to rename all text files, and this is really a standard task for an administrator. The previous suffix .txt should be changed to .old. Before doing this, a backup copy of the original should be stored in the archive directory. To make things even more exciting, files that are younger than the script you are using to carry out this process should be excluded. It works like this:

```
#!/bin/bash
 for i  i n * . txt
  do
   if [$i -ot $0 ]
   then
     cp $i archiv
     mv $i ''basename $i .txt'' .old
   fi
done
```

This time the for loop does not run through numerical values, but all files (in the current directory) that end with txt.

Inside the loop is a conditional statement that begins with if and ends with fi (the word is simply written backwards).

The condition itself is in square brackets and reads: "If $i is older than (ot stands for older than) $0". Here, $0 stands for the name of the current file (i.e. the one that is currently being called as a script). $1 is assigned the first parameter, $2 the second and so on.

Calling "meinSkript.sh Haus Maus 12" would result in the following assignment:

- `$0 → meinScript.sh`
- `$1 → Haus`
- `$2 → Maus`
- `$3 → 12`

In other words, the condition prevents files that are not older than the script file itself from being renamed (and copied).

You can find more of the numerous options that you can specify as a condition in the manual pages under man test. The shell also accepts the test command outside of `if`. The square brackets are a little easier to read in the condition and simply replace test.

Use `cp` (copy) to copy the `$i` file to the `archiv` directory. If this does not exist, `cp` assumes that `$i` should simply be copied to the file `archiv`. Although this does not produce an error, it is certainly not what you want: The next loop pass overwrites `archiv` again, and in the end you would only have a copy of the last file. It is therefore better to create `archiv` beforehand using `mkdir archiv` (make directory).The command mv (for move) is used for simple renaming. The first parameter is the old version, the second the new one.

The ***backticks*** (') in the second part are cunning little beasts. They allow you to execute another command in between and simply place its result in this position. The backticks can be found on your keyboard as “Accent Grave” (don't forget the shift key!), `basename` with two parameters is used to get the name of a file without path information. The second parameter also removes the specified suffix. `old` is simply added after the second backtick.

In effect, this script executes the following commands one after the other:

```
cp File1.txt archive
mv File1.txt File1.old
cp File10.txt archive
mv File10.txt File10.old
cp File2.txt archive
mv File2.txt File2.old
cp File3.txt archive
mv File3.txt File3.old
cp File4.txt archive
mv File4.txt File4.old
cp File5.txt archive
mv File5.txt File5.old
cp File6.txt archive
mv File6.txt File6.old
cp File7.txt archive
mv File7.txt File7.old
cp File8.txt archiv
mv File8.txt File8.old
cp File9.txt archive
mv File9.txt File9.old
```

Of course, it does not display them. At the end, the old files are in the directory and the originals are in the subdirectory `archiv`.

Not a bit cumbersome: AWK

As you can see, shell commands merge with other commands of your operating system during script programming. Sometimes the administrators themselves do not know whether a certain command is issued from the shell or whether it actually produces an external program call.

One example is `echo`. This quite simple instruction simply returns the parameters passed (to the standard output).

You can find more information on this in chapter 22 in the section “Standard channels”

Theoretically, this could be a program that calls the shell. In fact, it is a built-in command of the shell itself.

If you are interested in the topic in more detail, look under the term ***builtins*** (built-in commands) of the various shells.

If you want to program properly, you will quickly reach the limits of the shells. In particular, handling your own complex data structures is very tedious.

There are “real” scripting languages for this. A time-honored one, at least on UNIX systems, is called AKW, after the developers Aho, Weinberger and Kernighan.

You have read correctly, awk has nothing to do with awkward.

The program is used to structure the content of other files, and other requirements can also be handled with ease.

With AWK, for example, you can split each line of a file into individual words. With the `FS` parameter for Field Separator, you can also use any other character string as a separator. In addition, the program does not fidget if you use characters as numbers or vice versa. I also find the convention that the concatenation of character strings (i.e. the gluing of words into sentences) by ... nothing.

A simple space serves as a glue! What's more, searching for patterns in AWK is really, really easy ...

Suppose you have a small company and the names and sales of your sales representatives are in a text file. Each cell is structured in such a way that the target turnover appears after the name, followed by the turnover actually achieved (separated by spaces) and finally the commission to be granted - provided that the target was actually achieved.

The individual lines of this `provisionfile`could look like this:

- Linda 15000 21299 1000
- Hermann 25000 22824 3000
- Theodor 20000 24387 2000
-

If you now want to know how much commission you actually have to pay, you can write a small script.To do this, create a file called calculate_provision.awk:

```
#! /usr/bin/awk -f
BEGIN { sum = 0}
   { if ($2 < $3) sum += $4; }
END { print “Sum = ” sum }
```

The `awk` script must be supplied with your commission file from the shell, like this:

```
cat provisionfile I ./calculate_provision.awk
```

Don't forget to give `calculate_provision.awk` execution rights beforehand!

The script itself reads - almost - like a normal C program. The curly brackets combine blocks. The first block (after BEGIN), just like the last (after END), is only called once. The middle block, on the other hand, is called again for each line of the input file - i.e. for each sales representative.

$2 contains the promised turnover, $3 the actual turnover. Only if $2 < $3, i.e. if more turnover was generated than expected, is the amount agreed in $4 added to the total.

Incidentally, the sum=0 in the first block is superfluous. AWK automatically initializes all variables with zero and is no longer interested in the data type. I have inserted the command to increase the readability of the small script.

By the way, in the last line you can see what I have indicated regarding the concatenation of strings. The `print` automatically joins the two parameters and sticks them together to form a common string.

If you work with AWK long enough, you might lose the desire to program “properly” again at some point...

Diving for pearls with Perl

The Perl scripting language was originally invented by the American Larry Wall at the end of the 1980s to analyze log files. He continued the idea of scripting languages and gave his “pearl” the language scope of the Bourne shell, AWK plus components from C and other high-level languages.

As a linguist, Wall paid particular attention to ensuring that the language came closer to human intuition. He also made sure that a wide range of functions was possible with just a few commands. Incidentally, this also applies to all other scripting languages.

The name Perl was originally intended to be “Pearl”. However, there was already a programming language with the same name, so the “a” was simply dropped, making its meaning much easier to decipher for the German reader.

Incidentally, the name has nothing to do with the town of “Perl” in the border triangle of France, Luxembourg and Germany, opposite “Schengen”. The Schengen, by the way, to which the agreement of the same name owes its name.

With the advent of the WWW service on the Internet in the 1990s, Perl was mainly used to program websites. During this time, it dominated server-side scripting, i.e. programming on the web server side to prepare HTML code for the browser.

Take a look at the following Perl script, which gives you information about the type and version number of the browser used. You must place this script on the server side, grant it the necessary rights and allow the execution of Perl scripts:

```
#! /usr/bin/perl -w
use CGI;
$cgi = CGI->new;
print $cgi->header;
my $browser = $ENV{ '
HTTP_USER_AGENT' };
print «EndOfHTML;

<html> <head>
<title>Browser Information</title>
</head>
<body>
EndOfHTML

print "Your Browser is a $browser";
print "</body></html>";
```

You already know the first line. Yes, even `perl` is ultimately just an executable program that simply processes the instructions in its input (the contents of that very file) in sequence. The `-w` parameter also switches on warnings, which is very useful for debugging, especially at the beginning.

The use CGI indicates the use of the “Common Gateway Interface” library. There is now a vast number of such libraries in Perl. This scripting language is particularly strong in the area of string processing.

Variables are defined with a dollar sign at the beginning. A preceding my indicates a local use of the variable. This is where the linguist comes out a little when defining the language.

The end of the processing is also described in a flowery way as (die). Fortunately, you don't need that in this example.

`print $cgi->header`; outputs the HTTP header. Simple, isn't it?

There are numerous variables in the environment of the script. You can access these values, also known as environment variables, using `$ENV`. This has nothing to do with Perl directly. However, they also include information that your browser willingly transmits to the server. For example, its name, its version number, the operating system on which it is running, the screen resolution, the color depth, the size of the browser window, a list of all installed plug-ins and so on. Of course, this is also very useful. This allows the web server (or, for example, the Perl script on the server side) to respond precisely to the specific requirements of your browser and optimize the output accordingly.

The HTML header is described from the print “`EndOfHTML` to the line `EndOfHTML`. The closing body tag is specified at the very end. In between, you will simply find the content of the $browser variable with a small accompanying text. Try it out!

Perl shows particular strength when processing entire text files. ***Regular expressions (regex)*** play a particularly important role here. This is a kind of magic formula to achieve maximum effect with as little syntax as possible, see for yourself!

Suppose you want to format a given text a little more beautifully. For example, superfluous blank lines should be removed and words at the beginning of sentences should be capitalized automatically.

This is an easy exercise for Perl:

```
#!/usr/bin/perl
$sentence = “ this is a sentence, the   world is really great. “;
print “Sentence before: $sentence\n”;
$sentence =~s/\s+/ /g; # replace multiple whitespaces with a space
$sentence =~ s/ \s+//; # replace spaces at the beginning with nothing
```

```
$sentence s/(A[a-z])/\U\l/; # replace first character with capital letters
$sentence =~ s/(\. [a-z] )/\U\l/; # create capital letters at the beginning of
    the sentence
print “Sentence after: $sentence\n”;
```

At the beginning, you will find a nice mess in the $sentence variable: Lots of superfluous spaces and a horrible use of lowercase letters.

Use =~ to apply the right-hand side to the variable on the left. We do this a total of four times in the program. A typical regular expression looks like this: `s/…/…/…` The s stands for substitute. What is between the first two slashes (“/”) is replaced by what is between the second and third slash. At the end, after the third slash, there are some nice options.

The first replacement in the example searches for whitespaces with `\s`, i.e. spaces, paragraph marks and the like. The + means: one or more. Any number of whitespaces can therefore be replaced by a single space. The option `g` stands for globally and means: “as often as possible”. Otherwise, only the first occurrence would be replaced. After the first line, there is therefore exactly one space per whitespace.

Unfortunately, there is still a superfluous space at the very beginning. This is detected by the anchor A^. Just as ^ stands for the beginning of a line, $ means the end. This time the slashes 2 and 3 immediately follow each other. This means that the position found is not replaced by anything and is therefore removed.

The third replacement ensures that the text is capitalized at the beginning. To do this, simply search for any lowercase letter [a-z] at the beginning A. The square bracket indicates a set of letters, one of which must match.

This match is always saved under \1. Accordingly, further expressions on the left would be saved one after the other as \2, \3 and so on. This enables you to continue processing the passages that have already been found. In our case, this hit is replaced by its uppercase version (U stands for upper case) using \U.

You may already understand the fourth replacement on your own. The search pattern `\.` stands for a dot. A `.` alone, on the other hand, means any character. So you are looking for text patterns of the form:

- dot
- followed by a space
- followed by a lowercase letter

The latter is then replaced by its uppercase version. When you start the program, the following output is produced (Figure 48.1).

```
Sentence before: “ this is a sentence, the   world is really great.
Sentence after: This is a sentence. The world is really great.
```

Figure 48.1: Output of the Perl script

Impressed? Perl offers an almost absurd variety of possibilities for manipulating strings using regular expressions. You can find whole books on this. At this point, there is only enough space to give you a small foretaste.

The triumph of PHP

While Perl was originally invented to analyze log files, the Dane Rasmus Lerdorf developed the scripting language PHP (originally an acronym for “Personal Home Page”) directly for manipulating HTML code in the mid-1990s. The special feature of PHP is that you can integrate PHP scriptlets, i.e. small code components, into HTML very easily. In addition, PHP scripts can be found millions of times on the server side of websites.

Again, I can only demonstrate the power of this scripting language with a small example that will hopefully inspire you to further research in this direction.

A particular strength of PHP can be seen in the handling of web forms:

```
<html> <body>
<form method=“post” action=M < ?php echo $_SERVER[' PHP_SELF'];?>“>
Text: <input type=”textM name=“our_test”>
<input type=“submit”>
</form>

<?php
if ($_SERVER[“REQUEST_METHOD” ] == “POST”) {
Stext = $"REQUEST[‘our_test’ ];
    if(empty($text)) {
     echo “Please write a text!”;
    }else {
    echo strtoupper($text);
  }
 }
?)
<body><html>
```

Embedded between the body tags of an HTML page, you will find the PHP code here, which is framed by `<?php and ?>`. The action echo `$_SERVER [ ' PHP_SELF ' ]` is executed in the form area of the HTML page; the PHP script itself is executed within this very page. Using the HTTP command POST, the single form field (which I called `our_test` in the upper part) is simply output again to the normal variable $text using `$_REQUEST`. So that you can see that we can actually work with it like in any other scripting language, I have also specified as a condition what happens if the user simply leaves the field empty. In this case, the user is prompted: "Please write a text". Otherwise, the original text is simply output in capital letters (`strtoupper` for "String to upper case"). Not very creative, I admit, but hopefully still informative. You can see the result in Figure 48.2.

Figure 48.2: Calling a PHP script

JavaScript

In the second half of the 1990s, the JavaScript language began its triumphal march together with its relative, Java.

Despite their similar names, JavaScript and Java are fundamentally different languages. The former is a scripting language, while Java is an object-oriented high-level language with which you can create independent programs that are compiled beforehand.

The entire sixth part of your book is dedicated to the Java programming language!

Originally, JavaScript was mainly used on the client side. Browsers were supposed to be able to execute programmed code embedded in HTML.

To this day, there are still repeated attempts to manipulate the browser by executing malicious code in JavaScript. You should therefore always install the latest security patches when you activate JavaScript.

JavaScript is also increasingly used on the server side. Powerful libraries make it possible to map extensive functionality in the web environment with just a few lines of code.

As an example, here is a small program that calculates the body mass index client side using JavaScript:

```
<html/body>
</script>
</htmlx/body>
<form>
    Weight (in kg): <input id="weight" type="number">
    Height (in cm): <input id="height" type="number">
    <input type=button value="Calculate my BMI!"
       onClick="rechne()M >
</form>

<script>
function rechne() {
var height =
    document.getElementById("height")
    .value;
var weight = document.getElementById("weight").value;
var bmi = Math.round(weight*l00*100 /
    (height * height));
alert("Your BMI is " + bmi);
}
</script>
</htmlx/body>
```

Again, the entire code is located within the body tags of an HTML page. The first two fields of the form (`<form>`) are used to enter weight and height. Note that I have chosen number as the `type`. The input button is almost self-explanatory. The `onClick` event triggers the execution of the JavaScript function calculate ().

This in turn can be found within the following script tags. There, dl with function contains the definition of the function `calculate()` mentioned above. As you can see, the browser will first load the entire page before the function is executed. Therefore, the specification of `calculate ()` can also be placed after the form definition.

The values of size and weight are taken from the corresponding IDs of the document. The keyword `var` signals the definition of a variable.

The JavaScript object hierarchy

- The topmost object is the window.
- This contains the document.
- There can be various forms within a document
- These in turn contain the individual elements.

Document getElementByld, for example, calls a method at the document level. Other options would be document. title or document. URL to- At the window level, on the other hand, you can use, for example

- window. open() to open a new window,
- window.close() to close the current window,
- window. resizeTo() to adjust the size and
- window. moveTo() to move the current window

The calculation itself is done as weight in kilograms divided by height in meters squared. However, as I have requested the input in cm - to save the user decimal places, the height must be divided twice by one hundred, which corresponds to a respective multiplication in the numerator:

$$\frac{weight}{height/100 * height/100} = \frac{weight * 100 * 100}{height * height}$$

The rounding function Math. round() from the math library is used to truncate the decimal places. At the end, a new window is created using alert.

The alert() method could also have been called as a window or document method.

Figure 48.3 shows you an example of what can result from this.

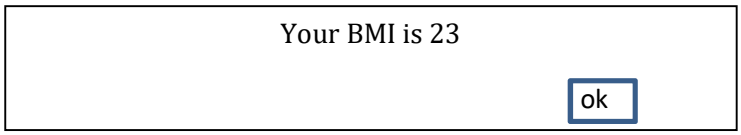


Figure 48.3: Execution of a JavaScript program

After several years of development, JavaScript has matured into a powerful, object-oriented scripting language. JavaScript is so much fun to use that even hard server elements that require high efficiency are now implemented in JavaScript here and there.

I would like to highlight the **Node.js** project as an example. This is an open source framework for a general network server for all possible platforms, written in JavaScript. The special features are the very flexible scalability. Fancy more?

You can find everything about the Node server and its functionalities at

```
https: //nodejs. org.
```

Don't forget the snake!

As an attentive reader, you may be thinking: “Since the Python programming language is interpreted, it should also be suitable as a scripting language!”

A very astute combination! Python is now also very popular in the field of scripting languages.

All the important basics about Python can be found in the seventh part of the book!

Using Python as a scripting language could hardly be easier. First, create a file that ends with `.py` by convention.

In the first line, call up your interpreter with the complete path, as you would for any other language. Then you can start working with Python code as normal. You have access to all the features of this language!

Since a whole section of this book is already dedicated to Python, a very small example will suffice here to show you how to interact with the operating system. To do this, import the library “os” (for operating system)

The small program reads the environment variables (with “`os.environ`”), which already represents a dictionary with all entries. Finally, the current shell is output as an example (which results in “`/bin/zsh`” on my system).

If you do not know the path to Python, this can be determined with the shell command `whereis python`

shell command!

```
#! /opt/anaconda3/bin/python

Import os

print(os.environ[“SHELL”])
```