Einführung in die Programmierung für Computerlinguisten mit Python

Max Hadersbeck et al.

Version 0.9.3

Einführung in die Programmierung für Computerlinguisten mit Python

Max Hadersbeck et al.

Vorlesungsskript
entwickelt am Centrum für Informations- und Sprachverarbeitung
der Ludwig–Maximilians–Universität

30. Januar 2017

Mitarbeiter am Skript:
Daniel Bruder, Yannick Kaiser, Dayyan Smith, Leonie Weißweiler

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Geschichte und Aufgaben der Informatik

Der Name Informatik besteht aus zwei Teilen:

- Information
- Mathematik

Die Informatik beschäftigt sich mit

- Informationssystemen:
  Struktur, Wirkungsweisen, Fähigkeiten und Konstruktionsprinzipien
- Modellen und Simulation
- Möglichkeiten der Strukturierung, Formalisierung und Mathematisierung von Anwendungsgebieten

Um für diese Bereiche Computerprogramme anbieten zu können, erforscht die Informatik:

- abstrakte Zeichen, Objekte und Begriffe
- formale Strukturen und ihre Transformation nach formalen Regeln
- die effektive Darstellung solcher Strukturen im Rechner und der automatischen Durchführung der Transformationen.

Anwendungen der Informatik finden wir unter anderem in folgenden Bereichen:

- Handel
- Industrie
- Wissenschaft
- Ingenieurwesen
- Sozialwissenschaft
- Medizin
- Ausbildung
- Kunst
- Freizeit

Eine sehr gute Einführung in die Informatik findet sich im Buch H.-P. Gumm, M. Sommer ({1994})

Ein sehr gutes Buch über die Thematik “Vom Problem zum Programm. Architektur und Bedeutung von Computerprogrammen” gibt es vom Tübinger Informatikprofessor Herbert Klaeren ({2001})

1.1 Rechner der nullten Generation

- *1623 – 1624*: Wilhelm Schickard
  Ein Universalgenie konstruiert die wohl erste Rechenmaschine. Auf sein Konto geht das Konstruktionsprinzip von Ziffernrädern und Zehnerübertragung.
- *1642*: Blaise PASCAL
  führt in Paris eine Maschine vor, die addieren und auf Umwegen auch subtrahieren kann. Sie hatte zehnstufige Zahnräder, der Zehnerübertrag wurde durch eine Klaue und Mitnehmerstifte realisiert.
- *1673*: Gottfried Leibnitz
  stellt in London seine Replica vor, die alle 4 Grundrechenarten mit 12-stelliger Anzeige bewältigen kann. Er erfindet außerdem das Dualsystem.
- *1822*: Charles Babbage
  Er entwickelt das Konzept der *difference engines* zur Überprüfung von Logarithmentafeln. Doch war es damals nicht möglich, ein mechanisches Getriebe zu bauen, bei dem 25 Zahnräder ineinandergreifen können, ohne zu verklemmen. 1833 entwarf er das Konzept der *analytical engine*, der ersten digitalen Rechenanlage. Sie enthielt schon sämtliche Komponenten einer modernen Rechenanlage:
  - arithmetischer Zahlenspeicher
  - arithmetische Recheneinheit
  - Steuereinheit zur Steuerung des Programmablaufs einschließlich der Rechenoperationen und des Datentransports
  - Geräte für Ein-/Ausgabe
  - dekadische Zahnräder für die Rechnung
  - zur Steuerung: Lochkarten
  - logische Verzweigungen und Entscheidungen
  - Die Addition sollte 1 sek., die Multiplikation zweier 50-stelliger Zahlen 1 Minute dauern.

  Zu Lebzeiten bezeichnete man ihn als Narren und keiner verstand ihn. Seine Frau Augusta Ada veröffentlichte seine Arbeiten und sein Sohn versuchte Teile dieser Maschine zu bauen, die heute noch im Science-Museum zu London zu besichtigen sind.
- *1847*: George Boole
  Er entwickelt einen Formalismus für die mathematische Behandlung von verschiedenen Aussageverknüpfungen, die mit den beiden Begriffen “wahr” und “falsch” arbeiten. Heute spricht man von der Bool´schen Algebra.
- *1938*: Konrad Zuse
  entwickelt ein Rechnerkonzept mit den logischen Grundoperationen. Er schlug einen programmgesteuerten Rechenautomaten mit 1500 Elektronenröhren vor. Die deutsche Regierung beurteilte eine solche Entwicklung als völlig sinnlos. Im Auftrag

der deutschen Luftfahrtindustrie entsteht 1941 die Z3, ein Rechner mit 600 Relais im Rechenwerk und 2000 Relais im Speicherwerk. In den USA wurde parallel dazu 1944 völlig unabhängig von Prof. H. Aiken, Professor für angewandte Mathematik an der Harvard University den Rechner MARK 1 entwickelt. Er bestand aus:

- 17 Gestellen mit Motoren und Drehschaltern
- 3000 Kugellagern
- 800 km Leitungsdraht
- 760.000 Einzelteilen

Eine Multiplikation benötigte 6 Sekunden (sie wurde in der NAVY für die Errechnung von Funktionstabellen verwendet).

- *1945*: John v. Neumann
  entwickelt in einer genialen Abstraktion des Steuerungsprozesses folgendes Programmierschema: Das steuernde Programm wird als Information codiert und in den Informationsspeicher des Rechners mit eingelagert.

  - Programmsprünge innerhalb des Programms möglich
  - Steigerung der Rechengeschwindigkeit

  Erst 1949 wurde dieses Konzept zum ersten Mal angewendet.

  Beispiel: Strickmaschine
  Fest eingestelltes Programm: Start → Ergebnis (keine Sprünge)
  Bei Neumann: **Programm + Daten**, Programm wird codiert → Sprünge sind erforderlich.

1.1.1 Arbeitsweise der Rechner der 0. Generation

Anweisungen, also Programme von außen mittels Lochkarten, Lochstreifen oder Programmstecktafeln.

Programmverzweigungen, Schleifen und Unterprogramme nur mit Spezialeinrichtungen wie zusätzlichen Programmstreifenleser und Endlosschleifen möglich.

1.2 Rechner der ersten Generation 1946 – 1965

- *1946*: wurde an der Universität von Pennsylvania der erste Röhrenrechner entwickelt →**ENIAC** **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer.

  - 18.000 Röhren
  - 1500 Relais
  - 70.000 Widerstände
  - 10.000 Kondensatoren
  - mehr als eine halbe Million Lötstellen

  - 30 Tonnen Gewicht
  - Addition $0.2 \cdot 10^{-3}$ Sekunden
  - Multiplikation $2.8 \cdot 10^{-3}$ Sekunden.
  - Leistungsverbrauch 150 KW
  - Rechensystem dezimal.

  Wurde für die Lösung von Differentialgleichungen am Ballistic Research Labor eingesetzt. Der Rechner war sehr zuverlässig, da die Röhren nur mit 25% ihrer Leistungsfähigkeit betrieben wurden. Die Ausfallrate betrug 2 - 3 Röhren pro Woche.

- *1951*: wurde das UNIVAC (**Univ**ersal-**A**utomatic **C**omputer)-System entwickelt:

  - Röhrenrechner
  - gemischtes Dual-Dezimal-System
  - Addition 120 ms
  - Multiplikation 1.8 ms
  - Division 3.6 ms

  Von der UNIVAC wurden insgesamt 45 Systeme hergestellt.
  Der Rechner bestand aus:

  - 5000 Röhren
  - 18.000 Dioden
  - 300 Relais
  - Als Speicher wurden Quecksilber-Verzögerungsleitungen (höchst giftig) verwendet.

- *1952*: entwickelt **IBM** einen legendären Rechner mit der Bezeichnung **701**
  Die Leistungsmerkmale sind:

  - sehr schnell, ebenfalls Röhrenrechner
  - als Hintergrundspeicher: Magnetband aus Kunststoff (bisher Metall-Magnetband).

- *1957*: Entwicklung der Programiersprache **FORTRAN** (**For**mula **Tran**slation)
  Das Einsatzgebiet der Programmiersprache war im Numerischen Bereich der Mathematik. Der Leiter der Forschungsgruppe, die FORTRAN entwickelte, war **John Backus**.

1.2.1 Anlagenstruktur der Rechner der ersten Generation

Die Rechner arbeiten im Blockzeitbetrieb.

Das Programm arbeitet mit einem Operator zusammen, der Schalter umlegen muss. Dadurch arbeitet das Programm direkt mit der Befehlsschnittstelle. E/A-Geräte werden ebenfalls vom Rechner bedient. Dadurch sehr langsamer Durchsatz.

1.3 Die Rechner der zweiten Generation 1957 – 1963

In dieser Rechnergeneration wurde folgendes entwickelt bzw. eingesetzt:

- Entwicklung der Transistortechnik
- Rechner arbeiten mit Binärsystem
- Magnetspeicher
- *1956*: IBM führt den Rechner 305 ein (CDC 66000).
- *1956*: Entwicklung der **IC-Technik** (integrated circuit) bei **Texas Instruments** Bei einem IC wird auf photomechanischem Weg in einem kleinen Halbleiterstück die elekronische Wirkung von Transistoren, elektrischen Widerständen und Kondensatoren erreicht. Die Anzahl der simulierten Bausteine resultiert aus der sogenannten Packungsdichte der Transistoren.
- *1958*: Entwicklung der Programmiersprache **LISP** von **John McCarthy**
- *1959*: Die Programmiersprache COBOL (**Co**mmon **B**usiness **O**riented **L**anguage) wird entwickelt.
- *1960*: In Europa wird die erste blockstrukturierte Programmiersprache ALGOL 60 (**Algo**rithmic **L**anguage) von führenden Wissenschaftlern der Welt auf mehreren Tagungen entwickelt! (unter Beteiligung von Prof. Bauer, TU München)

1.3.1 Anlagenstruktur der Rechner der zweiten Generation:

- Stapelbetrieb mit Vorrechnern
- Programme werden als Stapel eingegeben
- Langsame Peripherie auf Vorrechner
- Compiler, Bibliothek auf MB (Operator muss nur noch Schalter auf MB umlegen)

1.4 Rechner der dritten Generation 1964 – 1981

In dieser Rechnergeneration wurde folgendes entwickelt bzw. eingesetzt:

- IC-Technik
- Microprozessoren, 8 und 16 Bit
- Schnelle Speicher

Mikroprozessoren erweitern die bisherige Funktionalität eines ICs um folgende Komponenten, die auf photomechanischem Weg in einen Halbleiterbaustein eingebrannt werden. Microprozessoren bestehen aus folgenden Komponenten:

- Steuerwerk
- Rechenwerk

- Befehlszeilen
- Akkumulator
- Befehlsregister
- Register

Die einzelnen Komponenten konnten Binärzahlen verarbeiten. Die Wortlänge der Binärzahlen war in den ersten Microprozessoren 4 Bit.

- *1962*: Weiterentwicklung von FORTRAN zu **FORTRAN IV**
- *1965*: erster Prozessorrechner von DEC PDP8: Minicomputer, kostete weniger als 20.000$. Die Anzahl der Transistoren pro Chip nahm zu.
- *1968*: Es entsteht **ALGOL 68** und eine ANSI Spezifikation von **COBOL** **Co**mmon **B**usiness **O**riented **L**anguage. COBOL wird in der Betriebswirtschaft und dem Bankwesen eingesetzt, da COBOL sehr gute kaufmännische Routinen zur Verfügung stellt.
- *1970*: Erfindung des Mikroprozessors von **Intel** 1103 (4bit)
  Es beginnt die Arbeit an **PROLOG** = **pro**gramming in **log**ic.
  Weiterentwicklung von **LISP**.
  PASCAL wird von Niklaus Wirth erfunden
- *1971*: Der erste General-Purpose Microprocessor wird von der Firma **Intel** entwickelt:
  Intel 4004, 4 Bit, 108 kHz Taktung, Geschwindigkeit: 0,06 MIPS, enthält 2300 Transistoren. Der Chip wurde im Auftrag der japanischen Firma Busicom entwickelt und von der Firma Intel für $60000 zurückgekauft.
- *1972*: Epoche der Großintegration LSI: **L**arge **S**cale **I**ntegration (mehrere tausend Transistoren auf einem Chip), **Dennis Ritchie** entwickelt die Programmiersprache **C**
- *1974*: **Intel** entwickelt den ersten 8 Bit Chip, den **8080**. Er wurde als erster CHIP weitverbreitet in Ampelsteuerungen eingesetzt.
- *1975*: **Tiny BASIC** wird definiert und auf einem Microcomputer mit 2 KB RAM von **Bill Gates** und **Paul Allen** zum Einsatz gebracht. Die Entwicklung von **Kleinrechnern** beginnt, als komplette Leiterplatinen entwickelt wurden auf denen
  - ICs
  - Mikroprozessoren
  - Datenspeicher

  aufgelötet waren. Durch die Erweiterung der Platinen um Peripherie, Stromversorgung, Terminal, Keyboard und Gehäuse konnte man nun Kleincomputer selbst zusammenstellen.
- *1976*: Der 24-jährige **Steve Jobs** und der 20-jährige **Steve Wosniak** gründen ein Unternehmen und bauen um den Microprozessor **6502** den ersten Kleinrechner.

Sie vervollständigen die Kits um Peripherie (Tastatur, Bildschirm) und schreiben zugehörige Software. Sie nennen die Firma "**Apple**".
Die Packungsdichte der Transistoren vergrößert sich:

- *1977*: 16K-Speicher werden entwickelt.
  Die Chips haben 20.000 Transistoren auf einem Baustein.
- *1978*: Der zweite 8-bit-Microprozessor INTEL 8086, der Start der erfolgreichen x86 Architektur, wird entwickelt.
  Einführung des **ZILOG Z80** mit dem berühmten Betriebssystem **CP/M**
  Erste optische Datenspeicher werden vorgestellt (Phillips)
- *1979*: Höchstintegration auf Chips (VLSI: **V**ery **L**arge **S**cale **I**ntegration, d.h. mehr als 100.000 Transistoren auf einer Fläche von 20 x 40 mm^2)
  - neue Transistormaterialien: Galliumarsenid
  - schnelle Schalter: Josephson Schalter mit einer Schaltzeit von $13 \cdot 10^{-12}$ sec., allerdings nur bei einer Temperatur von -269C
  - große Speicherbausteine
  - 2-Mbit-Speicher (Bubble-Speicher). Als schnelle Hintergrundspeicher werden Plattenlaufwerke entwickelt. Programmiersprachen: PASCAL, LISP

1.4.1 Anlagenstruktur der Rechner der dritten Generation

Kleinrechner und Timesharingsysteme mit Bildschirm und Tastatur.

1.5 Die Rechner der vierten Generation 1982-1989

In dieser Rechnergeneration wurde folgendes entwickelt bzw. eingesetzt:

- 32-bit-Mikroprozessoren (160.000 Transistoren)
- 128k große Speicher auf einem Chip.
- Computer werden über Netzwerke verbunden. Damit beginnt das Zeitalter des Distributed Processing.
- *1983*: Die Programmiersprache **Smalltalk-80** wird implementiert.
  Die Programmiersprache **ADA** entsteht im Auftrag des DoD.
  ADA war der Versuch des Amerikanischen Verteidigungsministerium, eine Programmiersprache einzuführen, die als Standard für alle militärischen Entwicklungen verwendet werden sollte.

  Ein Microprocessor mit sehr guten I/O Möglichkeiten, der **MOTOROLA 68000**, wird entwickelt. Mit diesem Processor können die Graphische Oberflächen leicht programmiert werden. Er wird zum ersten Mal im Rechner **APPLE LISA** von Steve Woszniak und Steve Jobs eingesetzt.

- *1985*: **INTEL** entwickelt den 80386 Processor
- *1986*: Es entstehen TURBO PASCAL, Eiffel. wurde von **Bjarne Stroustrup** entwickelt und war eine Weiterentwicklung von "C", d.h. Abwärtskompatibilität zu C war vorhanden. Dazu kam das objektorientierte Konzept (programmierabhängige Definition von Objekten)
  Die Firma **MIPS** entwickelt den ersten **RISC** Prozessor **R2000**
- *1985*: Die Firma **SUN** entwickelt ihre RISC **SPARC** Architektur.
- *1988*: Es entsteht COMMON LISP und von Nikolaus Wirth das System OBERON, ein Nachfolger von **MODULA-2**
- *1990*: wird überarbeitet und es entsteht **Version 2.0**
- *1991*: Visual BASIC entsteht.
- *1995*: Die Programmiersprache ADA wird überarbeitet und von einem ANSI Komitee standardisiert zu **ADA95**

1.6 Die Rechner der fünften Generation

In dieser Rechnergeneration wurde folgendes entwickelt bzw. eingesetzt:

- größere Packungsdichte auf den Chips und dadurch größere und schnellere Speicher.
- Man spricht von 1 Million Transistoren auf einem Chip
- widearea Netzwerke
- künstliche Intelligenz und Expertensysteme
- VHLSI (very high large scale integration)
- 3D-Prozessoren
- selbstlernende Systeme (**A**rtificial **I**ntelligence)
- Software in Silicon
- natürliche sprachliche Systeme

Diese Rechnergeneration ist mit großen Ankündigungen gestartet und hat vieles, was sie erreichen wollte, nicht erlangt. Speziell in Japan wurden Milliardenbeträge in die Entwicklung von Systemen mit künstlicher Intelligenz gesteckt, die nicht zu dem erwarteten Ziel geführt haben. Folgende Vorhaben wurden nicht erreicht:

- "Der Computer, der denkt wie der Mensch",
- "Spracherkennung fließender Rede",
- "Automatische Übersetzung",
- "BIOChips",
- "Microprozessoren mit Lichtwellenleitern"

Dagegen haben sich in anderen Gebieten umwerfende Entwicklungen ereignet:

- Neue Mikroprozessortechnologie:
  - RISC Hardware statt CISC
  - General Purpose CPU's
  - Pipelineing und Parallelrechner
- Weltweite Vernetzung von Rechnern
  - Internet
- Multimediasysteme auf dem PC

1.7 Die Rechnergenerationen bis ca. 1990 im Überblick

Tabelle 1.1: Die Rechnergenerationen bis ca 1990 im Überblick

Gener.	**Jahr**	**Rechner**	**Eigenheiten**	**Software**	**Schnelligkeit**
Erste	1946 -65	ENIAC, UNIVAC, IBM 701	Röhrenrechner, Dezimalsystem	Gespeicherte Programme, Lochkarten	2 KByte Speicher 10 KIPS (Instruction/Sec)
Zweite	1957 -63	CDC 6600, IBM 306	Transistor-technik, Binärsystem	FORTRAN, ALGOL, COBOL	32 KByte Speicher 2000 KIPS
Dritte	1964 -81	IBM 360, PdP 11, CYBER 205	IC-Technik, Mikroprozessoren (4,8,16 Bit), LSI, Minicomputer, Magnetplatten	PASCAL, LIPS, Timesharing, Computergrafik	2 MByte Speicher 5 MIPS
Vierte	1982 -89	CRAY XMP, SPERRY 1100, Sparc (SUN)	Multiprozessoren, (32 Bit) VLSI, Microcomputer, Optische Platten	C, UNIX, ADA, PROLOG, Objekt-orientierte Sprachen, Datenbanken	8 MByte Speicher 30 MIPS
Fünfte	ab '89	CRAY T3D	RISC, Multipro-zessoren (64 Bit), Optische und neuronale Bausteine	Multimediale Betriebssys-teme, Expertensysteme	1 Gbyte Speicher Giga-Flops

1.8 Verwendete Abkürzungen

Gbytes

Integrated Circuit (= Transistor)

Large Scale Integration: mehr als 1000 Transistoren auf einem Baustein

Very Large Scale Integration: mehr als 100000 Transistoren auf einem Baustein

8 Bit, wird benötigt, um einem Buchstaben zu speichern.

1000 Byte,

100000 Byte, 1000 Kbyte,

10 hoch 9 Byte, 1000 Mbyte

Instructions per Second,

1 Million Instruktionen pro Sekunde

“Floating Point”-Operationen pro Sekunde

2 Die Programmiersprache Python

2.1 Allgemeines zu Python

> … I got tired of writing apps in C. … and I really didn't like Perl 3, which had just come out. So I decided to create my own language instead.

– Guido van Rossum, Erfinder von Python

> Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.
>
> The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF's mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF's home page is at https://www.python.org/psf/.
>
> Python allows you to split your program into modules that can be reused in other Python programs. The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for The Python Standard Library to get an idea of what's available. A wide variety of third-party extensions are also available. Consult the Python Package Index to find packages of interest to you.
>
> Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test

> functions during bottom-up program development. It is also a handy desk calculator.
>
> Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons: the high-level data types allow you to express complex operations in a single statement; statement grouping is done by indentation instead of beginning and ending brackets; no variable or argument declarations are necessary.
>
> Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.
>
> The latest Python source distribution is always available from python.org, at https://www.python.org/downloads/. The latest development sources can be obtained via anonymous Mercurial access at https://hg.python.org/cpython. The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.
>
> There are numerous tutorials and books available. The standard documentation includes The Python Tutorial.
>
> By the way, the language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with nasty reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

– Guido van Rossum

2.2 Pythonskript und Interaktiver Modus

Python kann auf eine von zwei Arten verwendet werden: Entweder als schnelles Tool in der Kommandozeile, oder indem man die Pythonanweisungen in einer Datei speichert und diese Datei vom Python-Interpreter ausführen lässt. Die Datei mit den Pythonanweisungen nennt man ein Pythonskript.

2.2.1 Python 2 vs Python 3

Es wird im Terminal zwischen den Kommandos `python` und `python3` unterschieden. `python` startet auf vielen Systemen Python **2**. Da dieses Skript mit Python **3** arbeitet, sollten Sie zum Starten von Python immer das Kommando `python3` verwenden! Es impfiehlt sich auf **jeden** Fall Python 3 zu verwenden, denn erst in Python 3 wird Unicode nativ unterstützt. Noch in Python 2 war ASCII der Standard!

2.2.2 Interaktiver Modus

Um den interaktiven Modus zu starten reicht es, in die Konsole `python3` einzugeben. Es öffnet sich die Eingabeaufforderung.

> Sie können diesen Modus jederzeit verlassen und zur Konsole zurückkehren, indem sie entweder `quit()` aufrufen, oder die Tastenkombination Strg-D ausführen.

Um mit Python `Hello World` auszugeben reicht schon eine Zeile `print("Hello World")`.

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

Der interaktive Modus eignet sich hervorragend, um unkompliziert Dinge auszuprobieren, und sogar als einfacher Taschenrechner ist er nützlich!

```
>>> 2 + 2
4
>>> 2 * (4 + 5)
18
```

Im Folgenden werden die meisten Beispiele im interaktiven Modus dargestellt werden, erkennbar an den `>>>`. Größere Programme und Hausaufgaben werden dagegen Python-Programme sein.

2.2.3 Python-Programme

Größere und komplexere Aufgaben werden sich nicht mehr ohne weiteres im interaktiven Modus erledigen lassen. Deswegen erlaubt es Python Skripte - sogenannte Module oder Programme - als Dateien anzulegen, und dann vom Python-Interpreter aufrufen zu lassen.

Eine Python-Datei ist eine Textdatei, normalerweise mit der Dateiendung .py.

2.2.4 Aufbau eines Python-Programms

Ein Python-Programm beginnt normalerweise mit der sogenannten *Shebang-Zeile* (siehe Start eines Python-Programms), gefolgt vom Inhalt des Programms. Das Programm wird vom Interpreter dann zeilenweise von oben nach unten ausgeführt.

```
#!/usr/local/bin/python3

print("Hello World")

x = 5
y = 2
z = x * y

print(z)
```

Die genaue Syntax von Python und was die einzelnen Statements bewirken werden Sie im laufe dieser Vorlesung lernen.

2.2.5 Start eines Python-Programms

Python-Programme sind keine Dateien in Maschinensprache, sondern Textdateien, die von Python interpretiert werden. Nachdem ein Python-Programm in einer Datei (z.B. `myprogram.py`) abgespeichert wurde, kann es mit dem Befehl `python3 myprogram.py` gestartet werden.

Damit der Befehl `python3 myprogram.py` auf einem Rechner funktioniert, müssen aber einige Bedingungen erfüllt sein:

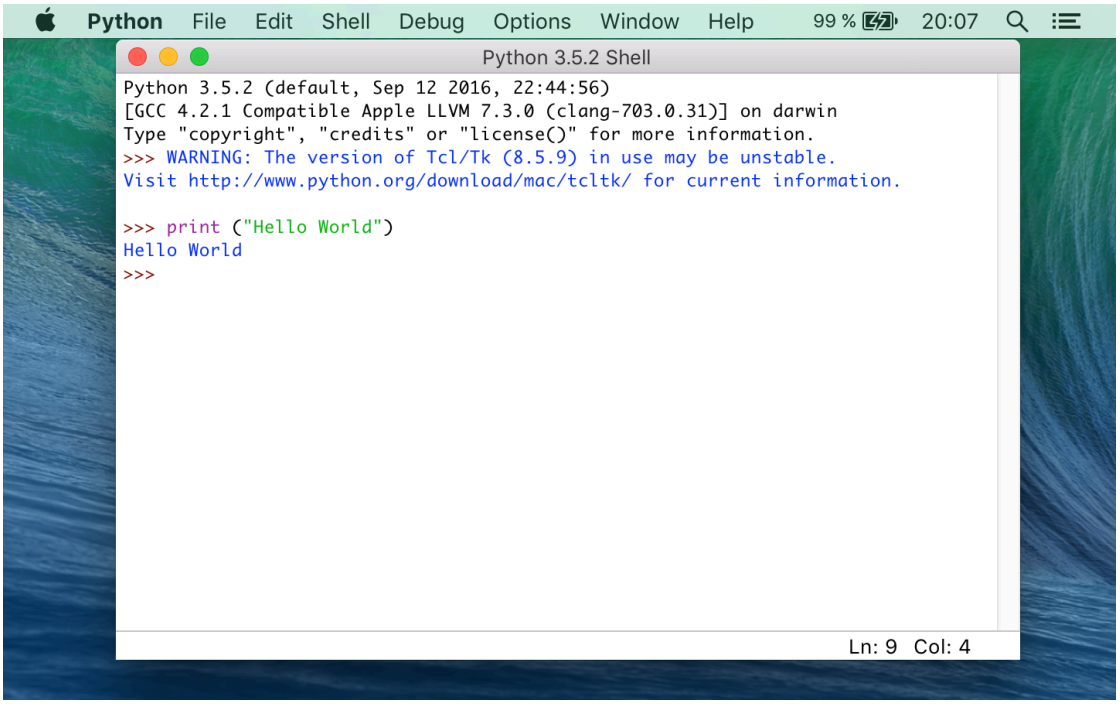
- Das Programm python3 ist auf dem Rechner installiert.
- Das Programm python3 ist in einem Verzeichnis installiert, welches in der PATH Umgebungsvariable hinterlegt ist. (Im allgemeinen `/usr/bin`, oder `/usr/local/bin`)

Um `myprogram.py` auch aufrufen zu können, ohne python3 davor zu setzen, müssen drei Bedingungen erfüllt sein:

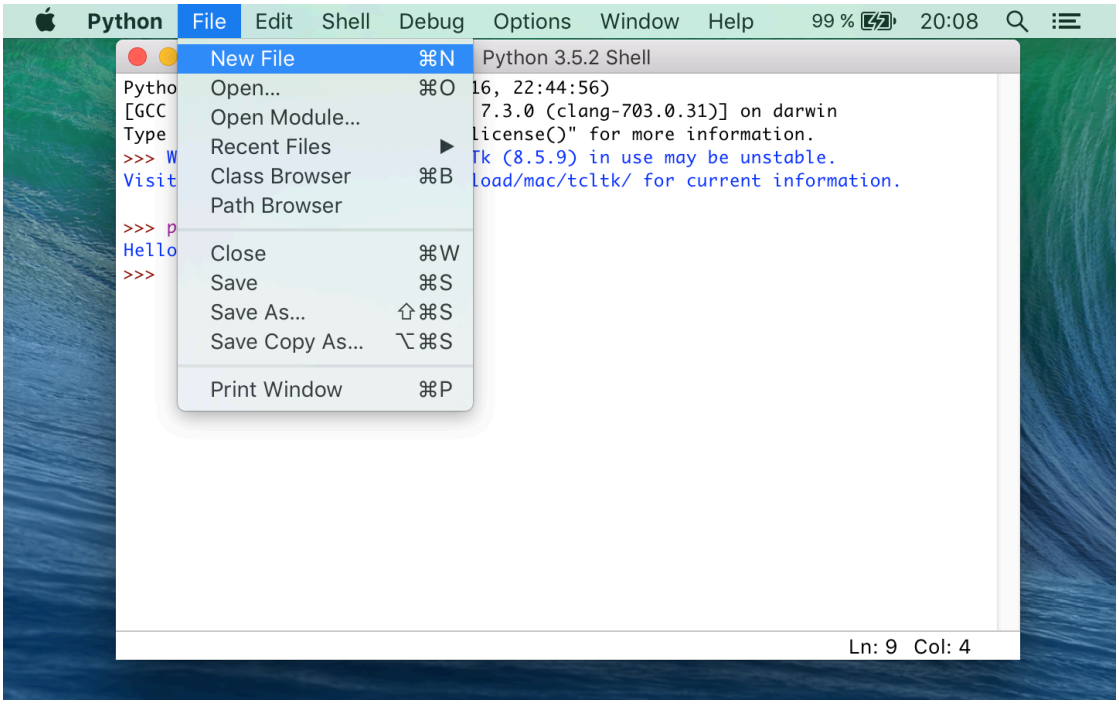
- Die Datei ist ausführbar (`chmod +x myprogram.py`)
- Die erste Zeile (= *Shebang-Zeile*) des Python Programms gibt an, wo python3 gefunden werden kann (z.B. `#!/usr/local/bin/python3`)
- Unter Linux gibt es den Befehl `env`, der den Installationsort einer Applikation ermittelt. Diesen Befehl kann man auch in der *Shebang-Zeile* verwenden. Dann lautet die erste Zeile des Pythonskripts: `#!/usr/bin/env python3`
- Das aktuelle Verzeichnis `.` ist im PATH

2.2.6 IDLE

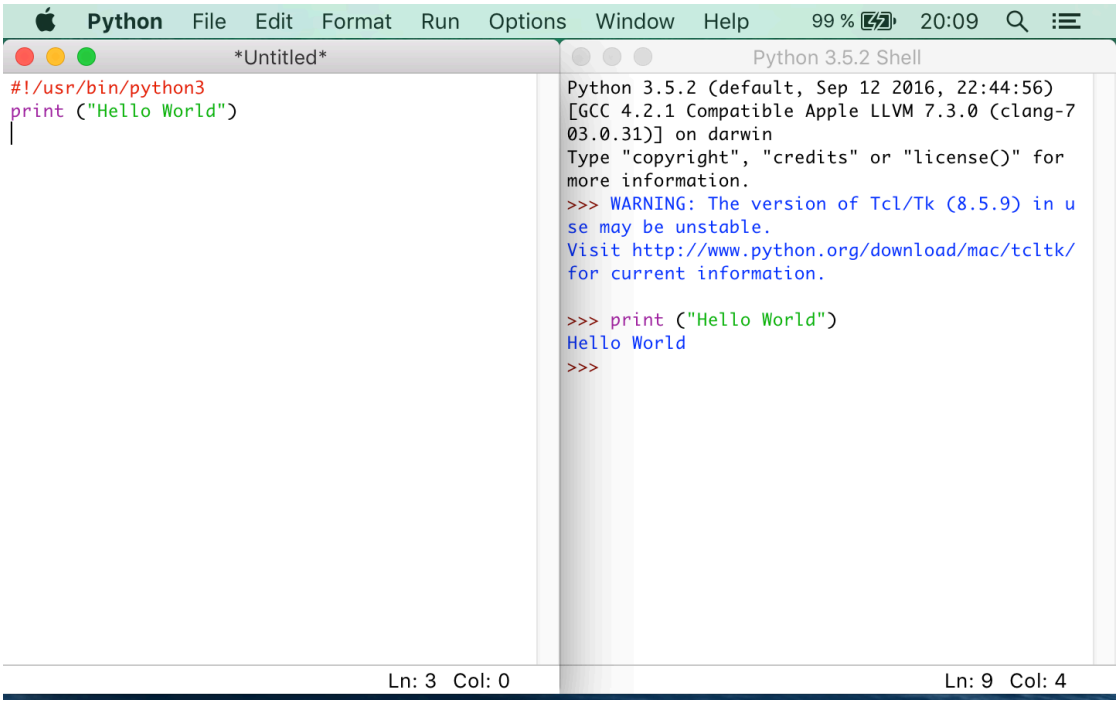
Ein praktischer Weg, Textdateien mit Python-Code zu erstellen und sofort auszuführen, ist die Pythonprogrammierumgebung IDLE, die mit dem bash Befehl `idle` bzw. auch `idle3` (für python3) gestartet werden kann.



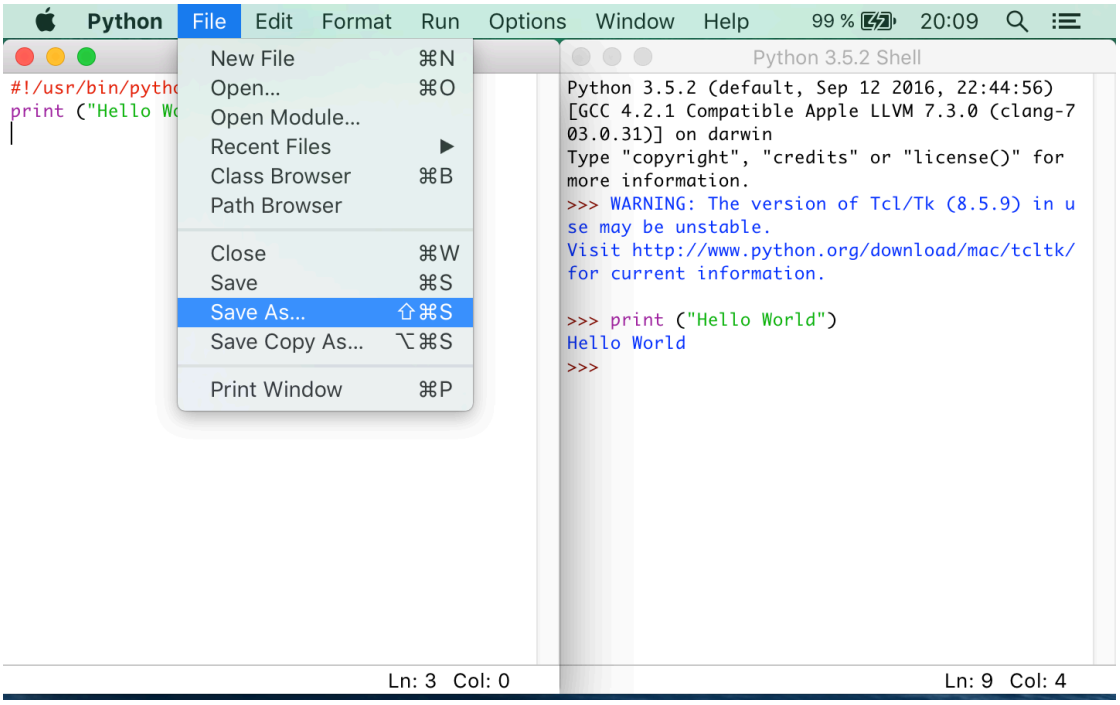
Es öffnet sich ein "Konsolfenster" auf dem Bildschirm, in dem automatisch python3 als `Python-Command-Shell` geöffnet ist, genauso als ob man in der bash Shell `python3` eingegeben hätte.



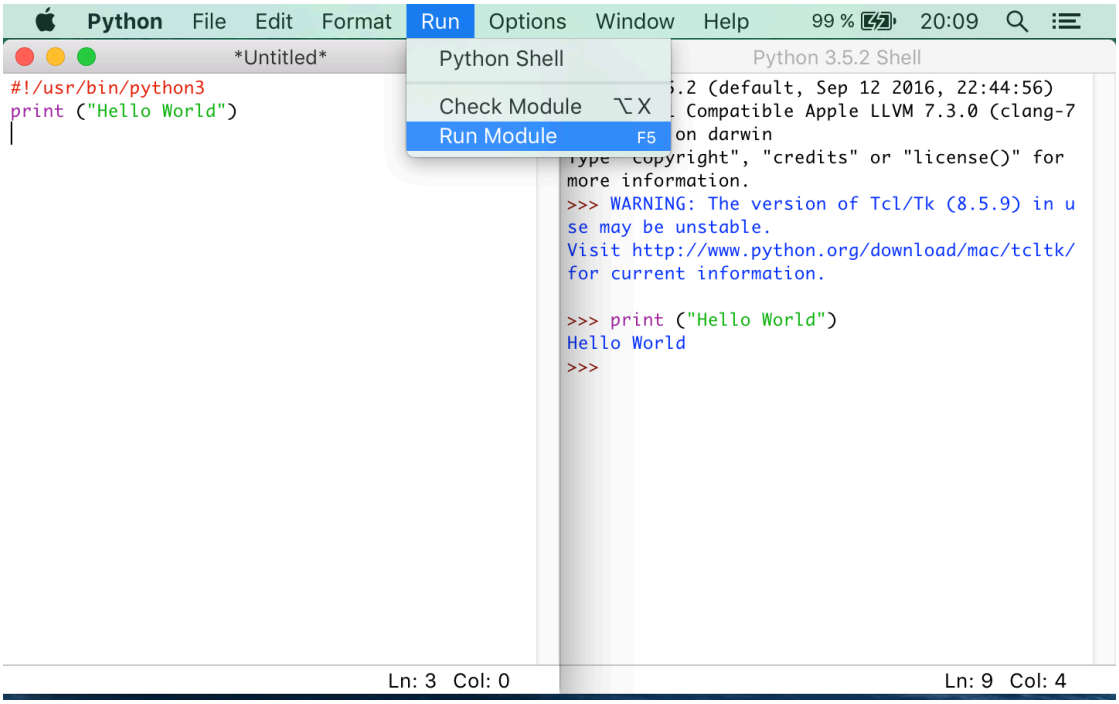
Neben vielen anderen Progammiertools, bietet die IDLE-Pythonprogrammierumgebung die Möglichkeit, auch Pythonskripte zu erstellen: Unter der Menuoption File -> New File kann man eine neue Datei erstellen.



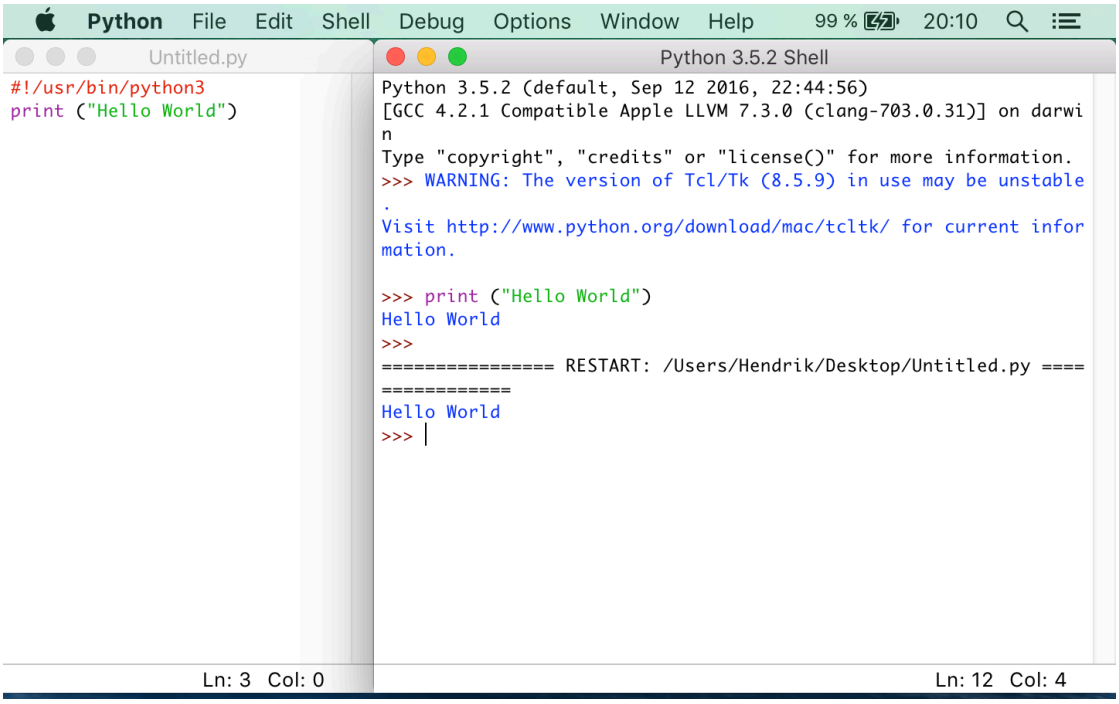
In der Datei gibt man seinen Python-Code ein...



...und speichert ihn unter file -> save as.



Jetzt kann man das Python-Programm entweder wie bisher von der Kommandozeile ausführen oder aber direkt in IDLE ausführen, unter run -> run module.



Das Ergebnis wird dann rechts in der in IDLE offenen Shell angezeigt.

2.3 Einfache Datenstrukturen: Wahrheitswerte, Zahlen, Strings und Listen

2.3.1 Vorbemerkung

Die Elemente eines Programms sollen das zu programmierende Sachgebiet so realistisch wie möglich wiederspiegeln. Deshalb wählt der Programmierer dementprechend die

- Datenstrukturen
- Operationen

aus einer Programmiersprache, die der Realität am nächsten kommen.

Die Datenstrukturen bestimmen die Art und Weise, wie Daten und Informationen in einem Programm verwendet werden können. Die wichtigsten Operationen innerhalb von Datenstrukturen sind:

- Zuweisung
- Vergleich

Im Folgenden werden die wichtigsten Datenstrukturen und die verfügbaren Operationen auf ihnen vorgestellt.

2.3.2 Wahrheitswerte

Eine Aussage hat den Wahrheitswert wahr (`True`), falls sie zutrifft, und falsch (`False`), falls sie nicht zutrifft. Bei der Auswertung von beliebigen logischen Aussagen hilft die Theorie der Aussagenlogik.

Sei $W(A)$ der Wahrheitswert der Aussage A:

$W(A) = True$, falls die Aussage A wahr ist

$W(A) = False$, falls die Aussage A falsch ist

Die beiden Wahrheitswerte `True` und `False` stellen den fest in Python eingebauten Datentypen `bool` dar, und müssen deswegen genau so geschrieben werden:

```
>>> True
True
>>> False
False
>>> type(True)  # type(x) gibt den Datentypen aus
<class 'bool'>  # 'True' ist vom Typen 'bool'
```

Logische Gleichheits- oder Verlgeichsoperationen werden seperat im Kapitel Logische Ausdrücke behandelt.

2.3.3 Zahlen

Python verwendet verschiedene Datentypen um verschiedene Arten von Zahlen zu speichern. Dabei wird im speziellen zwischen *Gleitkommazahlen* und *Ganzzahlen* unterschieden, da Gleitkommazahlen eine spezielle Kodierungsmethode im Arbeitsspeicher verwenden.

Datentyp	Beschreibung	Mutability
int	ganze Zahlen	immutable
float	Gleitkommazahlen	immutable

Integer sind Ganzzahlen, die intern als eine (beinahe) beliebig lange Bytefolge repräsentiert werden. Das hat zur Folge, dass anders als in vielen anderen Programmiersprachen - und anders als es noch in Python 2 war - keine Ober-/Untergrenze des Zahlenbereiches eines Integers besteht.

Eine Ganzzahl wird vom Interpreter als solche erkannt, wenn kein Komma verwendet wird:

```
>>> type(322)       # type(x) gibt den Datentypen aus
<class 'int'>       # '322' ist vom Typen 'int'
```

> Achtung: Sobald eine scheinbare Ganzzahl ein Komma enthält, wird sie als `float` interpretiert! 12.0 ist also eine Gleitkommazahl.

Float ist der Datentyp für Kommazahlen, dabei wird das Ergebnis von arithmetischen Operationen oftmals automatisch als ein `float` zurückgegeben!

Der Interpreter interpretiert eine Zahl immer dann als `float`, wenn ein Punkt verwendet wird. Es ist ebenfalls zulässig, die wissenschaftliche Schreibweise mit *Mantisse* und *Exponent* anzugeben:

```
>>> type(278.3)     # type(x) gibt den Datentypen aus
<class 'float'>     # '278.3' ist vom Typen 'float'
>>> type(2.783e2)
<class 'float'>
```

Weil der Speicherbereich für eine Zahl vom Datentypen `float` begrenzt ist, ist auch die Genauigkeit von Rechenoperationen mit `float`-Zahlen begrenzt. Wird eine Gleitkommazahl zu groß oder zu klein, so erhält sie den Wert `inf` oder `-inf` (“Infinity”). Dies ist kein Fehler - es kann aber unter Umständen nicht mehr sinnvoll mit so einer Zahl gerechnet werden:

```
>>> 1.3 + 1.6
2.9000000000000004
>>> 9.9e999
```

```
inf
>>> -9.9e999
-inf
>>> 9.9e999 - 42
nan     # "Not a Number" - Es kann von 'inf' nicht subtrahiert werden.
```

Die Rechenoperatoren für Zahlen werden im Kapitel “Arithmetische Operatoren” gesondert vorgestellt.

2.3.4 Strings

Strings, also Zeichenketten, werden in Python vom Datentyp `str` repräsentiert, Dabei werden seit Python 3 auch regionale Sonderzeichen problemlos verarbeitet.

Datentyp	Beschreibung	Mutability
str	Zeichketten	immutable

Strings werden dem Interpreter durch einfache oder doppelte Hochkommas gekennzeichnet:

```
"doppelt"
'einfach'
```

Darüber hinaus können *dreifache* einfache oder doppelte Hochkommata verwendet werden, wenn sich ein String über mehrere Zeilen der Quelldatei erstrecken soll:

```
"""Dies ist ein langer
String, welcher sich über mehrere
Zeilen erstrecken wird!"""
```

`str` ist, genau wie `list`, ein *sequenzieller* Datentyp. Deswegen sind alle Operationen, die für Listen gültig sind, auch für Strings gültig! Ein Wort ist nichts anderes, als eine Liste von Buchstaben, und genau als solche wird `str` von Python behandelt:

```
# Der String wird in der Variablen 'text' gespeichert
>>> text = "Der Sinn des Lebens"
>>> text[0]
'D'
```

2.3.5 Listen

Listen sind ein integraler Bestandteil jeder Form von Ordnung, und dürfen auch in Python nicht fehlen. Zuständig für Listen ist der Datentyp `list`, der eine sequenzielle Auflistung mit variabler Länge von beliebigen Objekten darstellt.

Datentyp	Beschreibung	Mutability
list	Zeichketten	mutable

Listen dürfen leer sein und werden initialisiert, indem die Bestandteile in eckigen Klammern angegeben werden `[]`:

```
[]
[1,2,3,4,5]
["es","war","einmal"]
[1,2,"Gitarre",["Liste","in","Liste"]]
```

> Formatierungskonvention: Vermeiden Sie unnötige Leerzeichen in der Listendefinition!

Wie zu sehen ist der Datentyp des Inhaltes einer Liste beliebig und kann innerhalb einer Liste frei gemischt werden. Der Grund dafür ist, dass in der Liste immer nur *Referenzen*, also Zeiger, auf die eigentlichen Objekte gespeichert werden.

Innerhalb der Liste kann jedes Element mit seinem *Index* angesprochen werden. Die Nummerierung beginnt dabei bei 0:

```
# Die Liste wird in der Variablen 'liste' gespeichert
>>> liste = ["erstes","zweites","drittes"]
>>> liste[0]
'erstes'
>>> liste[1]
'zweites'
>>> liste[2]
'drittes'
```

> Das letzte Element hat also den Index ‘Länge der Liste Minus Eins’

Die Operationen um Listen zu bearbeiten finden sich im Kapitel “List/Tuple-Operatoren”.

Eine Übersicht der Operationen für Strings findet sich im Kapitel “String-Operatoren”.

Slicing

Um auf die Elemente oder auf die Teile der Listen schnell zugreifen zu können, stehen in Python **Slice-Operatoren** zur Verfühgung.

```
liste[start:end] # Elemente vom 'start'-Index bis zum 'end'-Index
liste[start:]    # Elemente vom 'start'-Index bis zum Ende der Liste
liste[:end]      # Elemente von Anfang bis zum 'end'-Index
liste[:]         # eine Kopie von der ganzen Liste
list[start:end:step] # Elemente vom 'start'-Index
                     # bis zum 'end'-Index mit 'step'-Schritt
liste[-1]    # letztes Element der Liste
liste[-2:]   # zwei letzten Elemente der Liste
liste[:-2]   # alle Elemente der Liste außer der letzten zwei
```

Beispiel: Slicing in Kombination mit `del`-Anweisung

```
>>> liste = [1,2,3,4,5,6,7,8,9]
>>> del liste[1::2] # entfernt jedes zweite Element der Liste
[1,3,5,7,9]
```

List-Methoden

Hier sind die Methoden der Listen, die uns zur Verfühgung stehen, zusammengefasst.

```
list.append(x)
# Hängt ein neues Element x an das Ende der Liste an.

list.insert(i, x)
# Fügt ein Element x vor der gegebenen Position i ein.

list.remove(x)
# Entfernt ein Element x.

list.pop(i)
# Entfernt ein Element an der Position i.
list.pop()
# Entfernt das letzte Element der Liste.

list.index(x)
# Gibt den Index mit Wert x zurück.

list.count(x)
# Gibt zurück, wie oft x in der Liste vorkommt.

list.sort()
# Sortiert die Elemente in der Liste.

list.reverse()
# Kehrt die Reihenfolge der Listenelemente um.
```

List Comprehentions

'List Comprehentions' bieten einen prägnanten Weg eine Liste zu erstellen.

Zum Beispiel, wenn eine Liste von Suffixen der Länge 2 erstellt werden soll.

```
>>> suffixe = [word[-2:] for word in wordlist]
```

2.4 Variablen

Variablen sind eindeutige Namen, die stellvertretend für einen Wert oder eine Menge von Daten stehen. Anders als in der Mathematik ist die Variable nicht ein "unbekannter Wert", sondern hat immer einen eindeutigen und bekannten Inhalt, der sich im Laufe eines Programmablaufs beliebig oft ändern kann.

Die Variable steht dann zu jedem Zeitpunkt stellvertretend für den Bereich im Arbeitsspeicher des Computers, unter dem der aktuelle Wert der Daten gespeichert ist.

Als Variablennamen kann man beliebige Zeichenketten aus Buchstaben und Zahlen verwenden, wobei der Name mit einem Buchstaben (oder Unterstrich) beginnen muss und kein Leerzeichen enthalten darf. Nach dem ersten Zeichen dürfen auch Zahlen oder weitere Unterstriche verwendet werden. Variablen sind in Python Case Sensitive: Es werden Groß- und Kleinbuchstaben in unterschieden.

Beispiel: zulässige Variablennamen

```
a = 2
A = 3 # unterscheidet sich von a!
zahl = 1138
größteZahl = -1
neue_telefonnummer = 123987
wort1 = "python"
wort2 = "flying"
```

Beispiel: unzulässige Variablennamen

```
1wort       # beginnt mit einer Zahl
kleinste Zahl   # Leerzeichen
```

In Python muss bei einer Variablen, anders als in vielen anderen Programmiersprachen, nicht festgelegt werden, welchen Datentyp die Variable beinhalten soll. Welche Art von Daten sich hinter einer Variablen verbirgt wird automatisch erkannt und kann sich auch später im Programmablauf jederzeit ändern.

Für die Schreibweise von Variablen, Funktions- und Klassennamen gibt es eine von der Python Software Foundation herausgegebene Leitlinie:

Formatierungskonvention nach PEP 8:

> Vermeide die Variablennamen `I`, `O`, `l` – sie sind in manchen Schriftarten nicht von ähnlichen Zeichen zu unterscheiden!
>
> Konstanten sollten nur mit Großbuchstaben geschrieben werden: `ANZAHL_PLANETEN_IM_SONNENSYSTEM`
>
> Variablen/Bezeichner sollten im *lowercase* geschrieben werden: `verbleibende_versuche`
>
> Funktionsnamen sollten im *lowercase* geschrieben werden: `multipliziere_mit_zwei(...)`

Klassennamen sollten mit einem Großbuchstaben beginnen: `Meine_neue_Klasse`

Viele Programmierer benutzen für Variablen lieber den sogenannten *Camel-Case*: `verbleibendeVersuche`

Oberstes Gebot: Bleibe innerhalb eines Projektes einem Stil treu!

Die Tabelle *Schlüsselwörter in Python* zeigt, welche von Python reservierten Schlüsselwörter nicht als Variablen benutzt werden können:

Tabelle 2.4: Schlüsselwörter in Python

and	assert	break	class	continue	def	del
del	elif	else	except	exec	finally	for
from	global	if	import	in	is	lambda
not	or	pass	print	raise	return	try
while	with	yield				

In vielen englischsprachigen Quellen werden Variablen in Python *Identifiers* – Bezeichner – genannt.

2.5 Ausdrücke und Operatoren

Mit Hilfe von Operatoren können Operanden verknüpft werden, um neue Werte zu berechnen. Die Verknüpfung von ein oder mehreren Operanden mit einem oder mehreren Operatoren bezeichnet man Expression, oder auch Ausdruck. In Python gibt es arithmetische Ausdrücke, String-Ausdrücke und logische Ausdrücke. Arithmetische Ausdrücke kombinieren numerische Werte mit numerischen Operatoren, String-Ausdrücke verknüpfen Zeichen(ketten) mit Operatoren und logische Ausdrücke werden mit logischen oder Vergleichs-Operatoren gebildet. In Ausdrücken können auch Variablen vorkommen, die dann gemäß der Operatoren und des Variablentyps ausgewertet werden und deren Wert in die Berechnung des Ausdrucks einfließt.

Beispiel: Arithmetische Ausdrücke

```
2 + 3
10 / 2
1.5 * 5.75
200 - kleine_Zahl
(1 + a) * b
```

Beispiel: String-Ausdrücke

```
"Hallo" + " " + "Welt!"
'Ho' + 'ho' * 2
```

Beispiel: Logische Ausdrücke

```
hat_geklingelt or hat_geklopft
not (x > 2)
5 in [1,4,7,9]
```

2.5.1 Wertzuweisung

Jeder Variablen kann ein Wert über eine Konstante oder über einen Ausdruck zugewiesen werden. Der Typ des Wertes, der in einer Variable gespeichert wird, legt den internen Typen der Variable fest. Der Datentyp einer Variable kann durch Zuweisung eines Ausdrucks eines anderen Typen jederzeit geändert werden, was aus Gründen der Übersichtlichkeit aber nicht empfohlen wird!

> Eine Zuweisung erfolgt über das Gleichheitszeichen `=`
>
> Formatierungskonvention: Lassen Sie vor und nach dem Gleichheitszeichen ein Leerzeichen! `a = 1`

Zuweisung einer Konstanten

Variablen wird bei der Initialisierung oft eine Konstante, ein sogenannter Literal, zugewiesen. Als Konstante oder Literale werden feste Ausdrücke bezeichnet, die keine Variablen beinhalten.

Beispiel: Zuweisung von Konstanten

```
x = 5
y = -5
z = 1.8
mein_wort = "Monty"
weihnachten = '''Geschenke!'''
liste = ["rot","grün","blau"]
```

Zuweisung eines Ausdrucks

Variablen kann auch der Wert eines Ausdrucks zugewiesen werden. Hierbei wird zuerst der Ausdruck auf der rechten Seite des Gleichheitszeichens ausgewertet und das Ergebnis dann der Variablen zugewiesen.

> Die Richtung einer Zuweisung ist immer "Ziel = Ausdruck", also von Rechts nach Links
>
> Formatierungskonvention: Lassen Sie zwischen Rechenoperationen ein Leerzeichen, aber klammern sie bündig! `a = (1 + 2) * 3`

Beispiel: Zuweisung eines Ausdrucks

```
x = 1 + 2
zahl = (5.6 + 9) * mein_multiplikator
neues_wort = wort + wort2
wahrheitswert = zahl > 2
```

Multizuweisung

Python erlaubt Konstrukte, um schnell und einfach mehrere Variablen mit Werten zu belegen.

Es ist möglich, mehrere Variablen mit dem selben Wert zu belegen:

Beispiel: Multi-Zuweisung, Gleicher Wert

```
a = b = 1
wort1 = wort2 = "Gleichheit"
```

Außerdem ist es möglich, eine Liste von Werten auf mehrere Variablen "aufzuteilen":

Beispiel: Multi-Zuweisung, Sequenz

```
# Nach der Zuweisung ist a = 1, b = 2, c = 3
a, b, c = [1,2,3]
a, b, c = (1,2,3)
```

Um verschachtelte Werte auf diese Weise Variablen zuzuweisen, müssen diese entsprechend geklammert werden:

```
# Nach der Zuweisung ist a = 1, b = 2, c = 3
a, (b, c) = [1,[2,3]]
a, (b, c) = (1,(2,3))
a, (b, c) = [1,(2,3)]
```

2.5.2 Arithmetische Ausdrücke

Arithmetische Ausdrücke sind alle Ausdrücke, die ausschließlich Zahlen und zugehörige Operatoren verwenden. Variablen, in denen Zahlen gespeichert sind, dürfen Teil eines Arithmetischen Ausdrucks sein und werden verwendet, als wären sie Zahlen:

Beispiel: Arithmetischer Ausdruck

```
mein_ergebnis = 2 * (5 + meine_variable) ** 2
```

Arithmetische Operatoren

Die in Python verfügbaren arithmetischen Operatoren sind:

Tabelle 2.5: Arithmetische Operatoren in Python

Operator	Bezeichnung	Beispiel	Beschreibung oder Bemerkung
`+`	Addition	`10 + 20 = 30`	Addiert Werte
`-`	Subtraktion	`4.5 - 1 = 3.5`	Analog zur Addition
`*`	Multiplikation	`5 * 15 = 75`	Multipliziert
`/`	Division	`5 / 2 = 2.5`	Teilt die Werte, liefert eine Fließkommazahl zurück
`%`	Modulo	`11 % 5 = 1`	Liefert Rest der Division zurück
`**`	Potenzieren	`2 ** 10 = 1024`	Potenziert den linken mit dem rechten Wert
`//`	Floor	`9//2 = 4`	Teilt die Werte und rundet das Ergebnis auf die nächste Ganzzahl ab

Arithmetische Ausdrücke können – ganz, wie in der Mathematik – durch Klammern gruppiert werden:

```
1+2*3 = 7
1+(2*3) = 7
(1+2)*3 = 9
```

2.5.3 String-Ausdrücke

Zeichenketten verfügen über eigene Operatoren, mit denen sie konkateniert (vereint), oder wiederholt werden können. Außerdem ist jeder String in Python eine Liste von Zeichen, und kann als solche auch mit den Listenzugriffsoperatoren bearbeitet werden.

String, Tupel und Listen -Operatoren

Tabelle 2.6: String, Tupel und Listen -Operatoren

Operator	Bezeichnung	Beispiel	Beschreibung oder Bemerkung
`+`	Konkate-nation	`>>> "Ton" + "ne" 'Tonne'`	Addition und Konkatenation wird je nach Kontext unterschieden
`*`	Repetition	`>>> "La" * 5 'LaLaLaLaLa'`	Multiplikation und Repetition wird je nach Kontext unterschieden
`[i]`	Index	`>>> a = "Wald" >>> a[0] 'W'`	Liefert den Buchstaben an angegebener Stelle. Der erste Buchstabe hat den Index 0
`[i]`	Index	`>>> a = "Wald" >>> a[0] = "b" >>> print(a) 'bald'`	Setzt das Element beim angegebenen Index neu
`[-1]`	Index	`>>> a = "Wald" >>> print(a[-1]) 'd'`	Liefert das letzte Element
`[i:j]`	Ausschnitt	`>>> a = "Ansicht" >>> a[2:6] 'sich' >>> a = "Ansicht" >>> a[1:] 'nsicht' >>> a = "Ansicht" >>> a[:] 'Ansicht'`	Liefert den Bereich inklusive des linken, exklusive des rechten Index, genannt: “Slicing” Die Grenz-Indizes können weggelassen werden, zeigen Anfang/das Ende der Zeichenkette Logisch daraus folgend wird mit dieser Schreibweise eine exakte Kopie der Sequenz erzeugt
`[i:j:k]`	Ausschnitt	`>>> a = "Felder" >>> a[0:4:2] 'Fl'`	Liefert den angegebenen Bereich, wobei nur jedes k-te Element betrachtet wird

Tabelle 2.6: String, Tupel und Listen -Operatoren

`[j:i:-k]`	Ausschnitt	`>>> a = [1,2,3,4,5]`
		`>>> a[4:0:-1]`
		`[5,4,3,2]`
			Eine negative Schrittweite k und rückwärts gewählte indices lassen die Operation rückwärts laufen

`len(a)`	Länge des strings a	`>>> a="computer"` `>>> len(a) 8`	liefert die Anzahl der Buchstaben in einem String.

2.5.4 Logische Ausdrücke

Aussagen, denen man einen Wahrheitswert, nämlich 'wahr' oder 'falsch' zuordnen kann, bezeichnet man als logische Aussagen. Eine Aussage hat den Wahrheitswert wahr (`True`) falls sie zutrifft und falsch (`False`) falls sie nicht zutrifft. Bei der Auswertung von beliebigen logischen Aussagen hilft die Theorie der Aussagenlogik.

Sei W(A) der Wahrheitswert der Aussage A:

W(A) = True , falls die Aussage A wahr ist

W(A) = False , falls die Aussage A falsch ist

Die beiden Wahrheitswerte 'True' und 'False' stellen fest in Python eingebaute Datentypen dar, und müssen deswegen genau so geschrieben werden:

Beispiel: Direkte Zuweisung von Wahrheitswerten

```
var1 = True
var2 = False
```

Darüber hinaus weist Python auch jedem anderen Objekt einen Wahrheitswert zu. Jedes Objekt wird als 'True' angesehen, mit den folgenden Ausnahmen:

Beispiel: Objekte oder Werte, die als "False" interpretiert werden

```
None
False
0       # Die Null jedes numerischen Typen
[], (), ""  # Eine leere Sequenz
```

Logische Operatoren

Logische Ausdrücke können mit Hilfe von Logischen Operatoren aus Primitiven Objekten gebildet werden.

Logische Vergleichsoperatoren

Zum Vergleich von Zahlen und Buchstaben gibt es 6 Vergleichsoperatoren, die in den folgenden Tabellen gezeigt werden. Vergleichsoperatoren werden auch *Relationelle Operatoren* genannt.

Tabelle 2.7: Zahlen-Vergleichsoperatoren

Operator	Bezeichnung	Beispiel	Beschreibung oder Bemerkung
`==`	Gleich	`>>> 10 == 10 True`	Liefert 'True' genau dann, wenn beide Seiten des Ausdrucks logisch äquivalent sind
`!=`	Ungleich	`>>> 200 != 200 False`	Liefert 'True' genau dann, wenn die beiden Operanden nicht gleich sind
`>`	Größer	`>>> 1234 > 0 True`	Liefert 'True' genau dann, wenn der linke Operand größer als der rechte Operand ist
`<`	Kleiner	`>>> 123 < 0 False`	Liefert 'True' genau dann, wenn der linke Operand kleiner als der rechte Operand ist
`>=`	Größer oder Gleich	`>>> 42 >= 42 True`	Liefert 'True' genau dann, wenn der linke Operand gleich oder größer als der rechte Operand ist
`<=`	Kleiner oder Gleich	`>>> 42 <= 21 False`	Liefert 'True' genau dann, wenn der linke Operand gleich oder kleiner als der rechte Operand ist

Tabelle 2.8: String-Vergleichsoperatoren

Operator	Bezeichnung	Beispiel	Beschreibung oder Bemerkung
`==`	Gleich	`"python" == "python" True`	Liefert 'True' genau dann, wenn beide Seiten des Ausdrucks logisch äquivalent sind
`!=`	Ungleich	`"c++" != "c++" False`	Liefert 'True' genau dann, wenn die beiden Operanden nicht gleich sind
`>`	Größer	`"Zebra" > "Affe" True`	Liefert 'True' genau dann, wenn der linke Operand alphabetisch größer als der rechte Operand ist
`<`	Kleiner	`"Yacht" < "Beta" False`	Liefert 'True' genau dann, wenn der linke Operand alphabetisch kleiner als der rechte Operand ist
`>=`	Größer oder Gleich	`"CIS" >= "CIS" True`	Liefert 'True' genau dann, wenn der linke Operand alphabetisch gleich, oder größer als der rechte Operand ist
`<=`	Kleiner oder Gleich	`"Zimt" <= "Anis" False`	Liefert 'True' genau dann, wenn der linke Operand alphabetisch gleich oder kleiner als der rechte Operand ist

Diese 6 logischen Vergleichsoperatoren können auch auf alle anderen Standartdatentypen angewendet werden:

```
>>> [1,2,3] == [1,2,3]
True
>>> {"Name": "Peter"} == {"Name": "Peter"}
True
>>> [9] != ["haus"]
True
>>> {"Name": "Peter"} != {"Name": "Peter"}
False
>>> [9,9] > [2,2]
True
```

> Wenn Sequenzen verglichen werden, so werden die Elemente mit gleichem Index paarweise verglichen.

Logische Verknüpfungsoperatoren

Neben den Logischen Vergleichsoperatoren gibt es die Verknüpfungsoperatoren, die mehrere Wahrheitswerte oder Logische Ausdrücke zu einem neuen Wahrheitswert verknüpfen. Außerdem kann jede Logische Aussage negiert werden. Python benutzt an dieser Stelle vorgeschriebene Schlüsselwörter:

`and`, `or`, `not`

Wenn zwei Aussagen `A` und `B` verknüpft werden, ergibt sich folgende Wahrheitstabelle:

Aussage	Aussage	Logisches Und	Logisches Oder
A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Die Negation eines Wahrheitswertes liefert genau sein Gegenteil:

Aussage	Negation
A	not(A)
True	False
False	True

Beispiele für Logische Operatoren:

```
var1 = True
var2 = False

>>> var1 and var2
False
>>> var1 or var2
True
>>> not(var2)
True

>>> (2 == 2) and ("hallo" == "hallo")
True
>>> (not(2 < 1)) and (("Python" > "ABC") or (var1))
True
```

Logische Inhaltsoperatoren

Möchte man Herausfinden, ob bestimmte Elemente in einer Sequenz vorkommen, so ist das in Python ganz einfach: Die Schlüsselwörter

```
in, not in
```

liefern `True` bzw. `False`, je nachdem, ob sich das Element in der Sequenz befindet:

Beispiele: Logische Inhaltsoperatoren

```
>>> "a" not in "Gemüse"
True
>>> 3 in [1,2,3]
True
>>> "Python" in ["Monty","Grail","Frenchman"]
False
>>> (12,17) in [(1,3),(12,17),(45,45)]
True
>>> 3 not in [1,2,3]
False
```

Die Operatoren `in` und `not in` werden auch Membership Operators genannt.

Logische Identitätsoperatoren

`is, is not`

Die beiden logischen Identitätsoperatoren vergleichen, ob es sich bei zwei Variablen um Referenzen auf die selbe Instanz handelt. Dabei wird nicht verglichen, ob die *Werte* der Instanzen gleich sind, sondern, ob die *Referenzen* der Variablen auf die selbe Instanz zeigen.

Variablen mit *immutable* Datentypen referenzieren demzufolge ihrem Design nach immer die selbe Instanz, falls ihr Wert gleich ist, Variablen mit *mutable* Datentypen hingegen können sich auf verschiedene Instanzen dieses Typen beziehen:

Beispiel: `is`-Vergleich mit mutable-Datentyp

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> # mutable Daten => unterschiedliche Objekte und Adressen
>>> print(hex(id(a)))
0x7f575dd2ac88
>>> print(hex(id(b)))
0x7f575dd2a188
>>> a is b
False
>>> b = a
>>> a is b
True
>>> a is not b
False
```

Beispiel: `is`-Vergleich mit immutable-Datentyp

```
>>> a = 5
>>> b = 5
>>> # immutable Daten => selbe Objekte und Adressen
>>> print(hex(id(a)))
0x7f57639e7b20
>>> print(hex(id(b)))
0x7f57639e7b20
>>> a is b
True
>>> b = a
>>> a is b
True
```

Im besonderen der Vergleich `var is None` kann bedenkenlos verwendet werden, da `None` ein immutable Datentyp ist.

2.6 Kontrollstrukturen

Die Kontrollstrukturen kontrollieren den Ablauf des Programms. Sie verändern den Kontext von Variablen und Bestimmen die Reihenfolge der Anweisungen, die im Programm ausgeführt werden sollen.

In Python gibt es folgende Kontrollstrukturen:

- einfache Zuweisungen, Statements
- Blöcke
- Schleifen
  - while
  - for
- Fallunterscheidungen
  - if / elif / else
  - Bedingte Ausdrücke
- Funktionen

2.6.1 Blöcke

Blöcke dienen dazu, dem Python Interpreter anzuzeigen, welche Zeilen Code logisch zusammengehören. Dies zum Beispiel erforderlich, wenn es darum geht Schleifen, Fallunterscheidugen oder Funktionen zu schreiben. Erstere beide werdem im Folgenden in diesem Kapitel vorgestellt.

Ein Block besteht immer aus einer Reihe an Anweisungen, die alle die selbe Einrückung in der Datei oder im Prompt haben. Dabei wird empfohlen, pro Level der Einrückung 4 Leerzeichen zu verwenden:

```
Block1
Block1
Block1
    Block2
        Block3
        Block3
    Block2
    Block2
Block1
```

Werden innerhalb eines Blocks Leerzeichen und Tabs vermischt, so spuckt der Interpreter eine Fehlermeldung aus, im schlimmsten Fall kann sich sogar die Logik des Programmes ändern!

Die meisten Editoren bieten eine Einstellungsmöglichkeit an, um statt eines Tabs eine beliebige Anzahl an Leerzeichen zu setzen.

2.6.2 Fallunterscheidungen

Sehr häufig soll ein Programm nicht einfach nur eine Liste von Anweisungen abarbeiten, sondern in unterschiedlichen Situationen unterschiedliche Arbeitsschritte ausführen. Ein Passwort-Programm muss zum Beispiel unterscheiden, ob das eingegebene Passwort richtig oder falsch war.

Um diese Funktionalität zur Verfügung zu stellen bietet Python Fallunterscheidungen an, die über eine besondere Syntax definiert werden.

Die if-Anweisung

```
if Bedingung:
    Anweisungen
```

Bedingung

Die Bedingung muss ein logisch auswertbarer Ausdruck sein. Falls der Wert der Bedingung `True` ist (siehe: Wahrheitswerte) werden die Anweisungen innerhalb des Blocks ausgeführt. Ist der Wert der Bedingung `False`, dann werden die Anweisungen übersprungen und die Anweisungen nach dem Ende des Blocks werden sofort ausgeführt.

Anweisungen

Die Anweisungen stellen den *Körper* der `if`-Anweisung dar. Als solches müssen sie als ein *eingerückter Block* angegeben werden!

Falls die Anweisungen nur aus einer Anweisung bestehen, dann kann an Stelle des *Blocks* auch ein einzelnes Statement in der Zeile der Bedingung stehen.

Beispiel: `if`-Anweisung

```
if x == y:
    z = 5
    wort = "Homomorphismus"
    print ("z ist jetzt 5")
```

Beispiel: `if`-Anweisung mit einem einzelnen Statement

```
if x == y: print ("Erfolg!")
```

elif und else

```
if Bedingung:
    Anweisungen
elif: Bedingung:
    Anweisungen
```

```
elif: Bedingung
    Anweisungen
else:
    Anweisungen
```

Oft reicht eine einzige Fallunterscheidung nicht aus, zum Beispiel, wenn man eine Zahl auf mehrere Werte hin prüfen möchte. Um dies zu ermöglichen lässt sich ein `if`-Statement um beliebig viele `elif`-Statements erweitern, die der Reihe nach abgearbeitet werden.

> elif steht dabei für "else if"

Beispiel: `elif`-Statements

```
if zahl == 1:
    print("Eins!")
elif zahl == 2:
    print("Zwei!")
elif zahl == 3:
    print("Drei!")
```

Am Ende jeder `if-elif`-Kette lässt sich ein sogenanntes `else`-Statement anhängen, das immer dann betreten und ausgeführt wird, wenn *keine* der Bedingungen erfüllt worden ist, also die Bedingung jeder `if`- und jede `elif`-Anweisung `False` ergeben hat.

Beispiel: `else`-Statement

```
if zahl < 0:
    print("Zahl ist kleiner als Null!")
elif zahl == 0:
    print("Zahl ist genau Null!")
else:
    print("Zahl ist größer als Null!")
```

Eine `if`-Anweisung besteht aus

- genau einem `if`-Statment
- keinen oder beliebig vielen `elif`-Statements
- keinem oder einem `else`-Statement

Formatierungskonventionen:

- Die Anweisungen im Körper müssen gleich weit eingerückt sein. Die bevorzugte Methode sind 4 Leerzeichen pro Level
- Die Anweisungen stehen untereinander
- Jede Anweisung steht in einer eigenen Zeile

Bedingte Ausdrücke

```
A if Bedingung else B
```

Ein Bedingter Ausdruck ist eine einfache Fallunterscheidung, bei der der Ausdruck einen von zwei Werten annimmt, je nachdem, ob die Bedingung erfüllt ist, oder nicht. Man kann sich das so vorstellen, als ob der Ausdruck entweder durch A oder durch B ersetzt wird, je nachdem, ob die Bedingung erfüllt ist.

Beispiel: Zuweisung mit bedingtem Ausdruck

```
zahl = 5 if positiv == True else -5
# 'positiv == True' ist in diesem Fall die Bedingung
```

Beispiel: Bedingter Ausdruck im Mehodenaufruf

```
>>> x = 5
>>> print("x ist Fünf!" if x == 5 else "x ist ungleich Fünf...")
'x ist Fünf!'
```

> Wie zu sehen können Bedingte Ausdrücke dazu verwendet werden langen und womöglich umständlichen Code zu verkürzen, was allerdings zu lasten der Lesbarkeit geht. Gerade Programmier-Anfängern sei daher geraten, im Zweifel zu den einfacher lesbaren `if`-Anweisungen zu greifen.

2.6.3 Schleifen

Eine Schleife ermöglicht es einen Code-Block, den sogenannten *Schleifenkörper*, mehrmals hintereinander auszuführen.

Die while-Schleife

```
while Bedingung:
    Anweisungen
```

Bedingung

Die Bedingung muss ein auswertbarer logischer Ausdruck sein. Falls der Wert der Bedingung `True` (siehe: Wahrheitswerte) ist, werden die Anweisungen innerhalb des nachfolgenden Blocks solange ausgeführt, bis der Wert der Bedingung `False` ist.

Anweisungen

Die Anweisungen stellen den *Körper* der Schleife dar. Als solches müssen sie als ein *eingerückter Block* angegeben werden.

Beispiel: `while`-Schleife

```
>>> x = 0
>>> while(x < 5):
...     print(x)
...     x = x + 1
...
0       # Ausgabe
1
2
3
4       # Nach dem Schleifenaufruf: x gleich 5 -  Schleife wird abgebrochen.
```

Die for-Schleife

Eine weitere Methode eine Schleife zu realisieren ist die `for`-Schleife, die über eine Sequenz iteriert. Iterierbare Sequenzen, die bereits bekannt sind, sind Listen und Strings, später werden noch Tupel und Dictionarys hinzukommen.

```
for Variable in Sequenz:
    Anweisungen
```

Sequenz

Die Sequenz, die durchlaufen werden soll, muss ein sogenanntes *iterierbares Objekt* sein, dessen Elemente als eine Sequenz abgerufen werden können. Solche Datentypen sind unter Anderem:

- `list` (Liste)
- `str` (String)
- `range` (Funktion)

Variable

Als Zählvariable wird eine *neue* Variable deklariert, die bei jedem Schleifendurchgang das aktuelle Element der Sequenz beinhaltet. Achtung: Wird als Zählvariable eine Variable verwendet, die vor der Schleife schon benutzt wurde, so wird ihr Wert überschrieben!

Beispiel: `for`-Schleife über eine Liste

```
>>> for zahl in [1,2,3]:
...     print(zahl)
...
1
2
3
>>> x = ["Eine","Deine","Seine"]
```

```
>>> for element in x:
...     print(wort + "s")
...
'Eines'
'Deines'
'Seines'
>>> wort = "CIS"            # Ein String ist eine Sequenz von Buchstaben
>>> for b in wort:
...     print(b)
...
C
I
S
```

Die range-Funktion

In vielen Fällen möchte man einfach nur einen Codeblock eine bestimmte Anzahl von Durchläufen ausführen lassen. Um dies einfach zu bewerkstelligen gibt es die eingebaute `range`-Funktion, die eine Liste von Zahlen erzeugt:

Die `range`-Funktion kann auf drei Arten aufgerufen werden.

```
range(stop)
range(start,stop)
range(start,stop,schritt)
```

- Mit *einem* Argument ("stop") wird von 0 bis *ausschließlich* dem Wert des Argumentes gezählt
- Mit *zwei* Argumenten ("start","stop") wird von *einschließlich* dem Wert des ersten Argumentes bis *ausschließlich* dem Wert des zweiten Argumentes gezählt
- Wird ein *drittes* Argument angegeben ("schritt"), so wird statt in Einer-Schritten in Schritten gezählt, die diesem Argument entsprechen
- Wird das *dritte* Argument als *negative* Zahl angegeben, so wird herabgezählt
- Alle Argumente müssen *ganze Zahlen* sein

Beispiel: Benutzung der `for`-Schleife als Zählschleife

```
>>> for x in range(4):      # 0 bis ausschließlich 4
...     print(x)
...
0
1
2
3
```

```
>>> for x in range(2,4):    # 2 bis ausschließlich 4
...     print(x)
...
2
3
>>> for x in range(4,10,2): # 4 bis ausschließlich 10, 2 pro Schritt
...     print(x)
...
4
6
8
>>> for x in range(10,5,-1):    # 10 bis ausschließlich 5, negativer Schritt
...     print(x)
...
10
9
8
7
6
```

Abbruch und Unterbrechung einer Schleife

> Vorbemerkung: In den allermeisten Fällen können die folgenden Schlüsselwörter komplett vermieden werden, in dem die zugrundeliegende Problematik in die Bedingung der Schleife oder in eine Fallunterscheidung verlagert wird. Manchmal erhöhen diese Statements aber die Lesbarkeit des Programmes oder sparen Variablen ein

Zur vorzeitigen Unterbrechung eines Schleifendurchlaufes, oder zum vorzeitigen Abbruch der gesamten Schleife, stellt Python zwei Schlüsselwörter zur Verfügung:

> `break` und `continue`

`break` beendet die Schleife sofort, das Programm fährt nach dem Ende des Schleifenkörpers fort

`continue` beendet den aktuellen Schleifendurchgang ab und fährt sofort mit dem nächsten Schleifendurchgang fort. Im Falle einer `for`-Schleife erhält die Zählvariable den nächsten Wert.

Beispiel: `break`-Statement

```
# Programm:
# Es soll geprüft werden, ob eine sehr lange Liste die Zahl 5 enthält
treffer = False
liste = [12,7,5,9,22,8]     # Stellvertretend für die lange Liste
```

```
for x in liste:
    if x == 5:
        treffer = True
        break           # Beendet die Schleife bei einem Treffer
if treffer:
    print("5 gefunden!")
else:
    print("Fehlschlag!")
# Ausgabe:
'5 gefunden!'
```

Beispiel: `continue`-Statement

```
# Programm:
# Es sollen in einer Liste alle Zahlen, die
# ungleich 0 sind, mittels Division verarbeitet werden
liste = [235,0,532,78,1004]
for x in liste:
    if x == 0:          # Ist die Zahl gleich 0...
        continue        # ...wird sie übersprungen,
                                # da sonst durch 0 geteilt werden würde!
    print(100/x)        # Alle anderen Zahlen werden normal behandelt
# Ausgabe:
0.425531914893617
0.18796992481203006
1.2820512820512822
0.099601593625498
```

2.7 Eingabe und Ausgabe auf die Konsole

2.7.1 Die input-Funktion

Zum Einlesen von User-Input von der Konsole gibt es in Python die besondere integrierte Funktion `input()`. Wird diese Funktion in einem Python-Programm aufgerufen, so pausiert das Programm so lange bis der Nutzer eine Eingabe gemacht hat und diese durch Drücken der Eingabetaste beendet hat. Die Eingabe wird von der Funktion dann als String an das Programm übergeben und kann zum Beispiel an eine Variable zugewiesen werden.

Folgendes Programm liest vom Terminal einen String und speichert ihn unter der Variable `passwort` ab:

```
#!usr/local/bin/python3

print("Bitte geben sie das Passwort ein: ")
passwort = input()
print("Die Eingabe war " + passwort)
```

Die Funktion `input` erlaubt es, einen String als Argument anzugeben:

```
input("Text")
```

Dieser Text wird dem Benutzer dann als Eingabeaufforderung angezigt. Man kann das obrige Programm also folgendermaßen abkürzen:

```
#!usr/local/bin/python3

passwort = input("Bitte geben sie das Passwort ein: ")
print("Die Eingabe war " + passwort)
```

Möchten Sie etwas anderes als Strings einlesen, zum Beispiel das Alter des Benutzers erfragen (Integer), so muss das Ergebnis des Aufrufs von `input` in den jeweiligen Datentypen konvertiert werden. Die Funktion `input` selber liefert immer nur einen String als Ergebnis.

Glücklicherweise bietet Python für jeden Standard-Datentypen eine integrierte Funktion, die versucht, ihr Argument in den jeweiligen Typen umzurechnen. Unter Anderem:

Name	Datentyp	Funktion	Bemerkung
Integer	`int`	int(x)	
Gleitkommazahl	`float`	float(x)	
Liste	`list`	list(sequenz)	Kann nur sequenzielle Objekte umrechnen, z.B. Strings: `list("ab")` zu `['a', 'b']`

Ein Programm, das nach dem Alter des Nutzers fragt, könnte also etwa so aussehen:

```
#!usr/bin/local/python3

input_string = input("Bitte geben Sie ihr Alter an: ")
alter = int(input_string)

if alter >= 18:
    print("In Ordnung, Sie dürfen passieren.")
else:
    print("Verzeihen Sie, erst ab 18!")
```

Da man Funktionen auch verschachteln kann, lässt sich das Alter-Beispiel auch vereinfachen:

```
alter = int(input("Bitte geben Sie ihr Alter an: "))
```

2.7.2 Die print-Funktion

Wir haben die `print`-Funktion schon oft verwendet, doch sie soll in diesem Zusammenhang noch einmal gesondert behandelt werden. Die integrierte Funktion `print` gibt ihre Argumente der Reihe nach auf der Standard-Ausgabe aus.

Die Standard-Ausgabe ist per Default die Konsole

```
>>> print("Eins", "Zwei", "Drei")
Eins Zwei Drei
```

Wird `print` ein komplexer Datentyp übergeben, so versucht `print` dieses Objekt in einen lesbaren String umzuwandeln und auszugeben:

```
>>> print(["Franz","Annegret","Herbert","Susanne"])
['Franz', 'Annegret', 'Herbert', 'Susanne']
```

2.8 Lesen und Schreiben von Dateien

Zur permanenten Speicherung von Daten verwendet man in Betriebssystemen Dateien. Dateien werden über einen Namen innerhalb eines Verzeichnisses (Ordners) eindeutig identifiziert. Unter UNIX werden Dateien, die binäre Dateien speichern, Binärdateien und Dateien, die Texte speichern, Textdateien genannt. Unter UNIX wird eine Textdatei als geordnete Folge oder "stream" von Zeichen betrachtet. Die einzelnen Zeilen sind durch ein Zeilentrennzeichen (Newline, `\n`) getrennt.

Folgende 5 Zeilen werden mit einem Texteditor in die neue Datei file.txt geschrieben:

```
Eins
Zwei
Drei
4 5 6 7 8
Ende
```

Damit die Information des Zeilenumbruchs nicht verloren geht, wird in der Datei an die Stelle des Zeilenumbruchs ein sogenannter Special Character eingefügt. Dieser Special Character hat die Aufgabe, bei der Ausgabe einen Zeilenumbruch zu bewirken. Der Special Character hat bei UNIX den Octal Code 12 und heißt *Newline*. Aus diesen Gründen wird auf der Festplatte der Inhalt folgendermaßen Zeichen für Zeichen abgespeichert:

```
E i n s nl Z w e i nl D r e i nl 4 sp 5 sp 6 sp 7 sp 8 nl E n d e
```

wobei `sp` in diesem Beispiel die Kodierung für Space und `nl` die Kodierung für Newline ist.

Unter UNIX gibt es den Befehl `od` (Octal Dump) mit dem man den Inhalt einer Datei Byte für Byte darstellen ("dumpen") kann. Dem Befehl muss man angeben, mit welchem Format die Zeichen der Datei konvertiert werden sollen, bevor sie auf dem Terminal ausgegeben werden. In der linken Spalte gibt der Befehl `od` die oktale Bytenummer des ersten Zeichens der Datei aus, in den restlichen Spalten die Bytes entsprechend der ausgewählten Konvertierung. (Weitere Information: `man od`)

Ausgabe der Zeichen als Oktalwert

```
$ od -b file.txt
0000000 105 151 156 163 012 132 167 145 151 012 104 162 145 151 012 064
0000020 040 065 040 066 040 067 040 070 012 105 156 144 145
0000036
```

Ausgabe der Zeichen als ASCII Wert

```
$ od -a file.txt
0000000    E   i   n   s  nl   Z   w   e   i  nl   D   r   e   i  nl   4
0000020   sp   5  sp   6  sp   7  sp   8  nl   E   n   d   e
0000035
```

Befehle zur Ausgabe von Textdateien interpretieren die Newline-Zeichen (abgekürzt: nl) bei der Ausgabe auf das Terminal richtig und erzeugen einen Zeilenumbruch anstelle z.B. der Buchstaben nl.

Beispiel: Textbefehl `cat` zur Ausgabe einer Textdatei

```
$ cat file.txt
```

```
Eins
Zwei
Drei
4 5 6 7 8
Ende
```

2.8.1 `open` und `with` statement

Möchte man den Inhalt der Datei mit einem Python Programm lesen, dann muss im Python Programm ein file-Objekt erzeugt werden. Dieses wird oft einer Variable zugewiesen - im unten stehenden Beispiel die Variable `f` (file). Das File-Objekt kann man sich vorstellen wie einen Verweis auf die Datei im Dateisystem des Rechners. Der `open`-Befehl legt diesen Verweis an und öffnet die Datei. Dabei wird aber noch nicht der ganze Inhalt gelesen oder ähnliches: Was mit dieser Datei passieren soll, entscheidet der Programmierer:

```
>>> f = open('file.txt')
>>> f = open('file.txt', 'r')
>>> f = open('file.txt', 'w')
```

Standardmäßig wird eine Datei zum Lesen geöffnet, sie kann aber auch in anderen Modi geöffnet werden:

Tabelle 2.12: Lese- und Schreibmodi

Modus	File Pointer	Beschreibung
r	Anfang	Öffnet die Datei ausschließlich zum Lesen. Der File Pointer zeigt auf den Anfang der Datei.
r+	Anfang	Öffnet die Datei zum Lesen und Schreiben. Der File Pointer zeigt auf den Anfang der Datei.
w	Anfang	Öffnet die Datei ausschließlich zum Schreiben. Überschreibt die Datei, falls sie bereits existiert, sonst wird eine neue erstellt.
w+	Anfang	Öffnet die Datei zum Lesen und Schreiben.

Modus	File Pointer	Beschreibung
		Überschreibt die Datei, falls sie bereits existiert, sonst wird eine neue erstellt.
a	Ende	Öffnet die Datei zum Anhängen, der File Pointer zeigt also auf das Ende der Datei, falls diese bereits existiert. Existiert die Datei noch nicht, wird eine neue erstellt.
a+	Ende	Öffnet die Datei zum Lesen und Anhängen, also im `a` Modus, mit der zusätzlichen Möglichkeit zu lesen.

Öffnet man eine Datei mit dem `open` statement, so muss diese nach Benutzung mit `close` wieder explizit geschossen werden. Passiert dies nicht, so kann es zu Fehlern im Dateisystem kommen, z.B. wenn das Programm abstürzt oder ein anderes Programm auf die gleiche Datei zugreifen möchte.

```
>>> f = open('file.txt')
>>> # do stuff with file
...
>>> f.close()
```

```
>>> f = open('file.txt','a')
>>> # do stuff with file
...
>>> f.close()
```

Eine andere Möglichkeit Dateien zu öffnen ist es das `with`-Statement zu benutzen.

```
>>> with open('file.txt','r') as f:
...     # do stuff with file
...     f.readlines()
...
['Eins\n', 'Zwei\n', 'Drei\n', '4 5 6 7 8\n', 'Ende\n']
```

Mit diesem Statement existiert das File-Objekt nur innerhalb des `with`-Blocks. Damit ist sichergestellt, dass die Datei geschlossen wird, nachdem der `with`-Block zu Ende ist: Beim Verlassen des `with`-Blocks wird die Datei von Python automatisch geschlossen.

2.8.2 Zeilenweises Lesen

Nachdem eine (Text-)Datei geöffnet worden ist, möchte man diese oft zeilenweise lesen. Die Funktion `readline` gibt bei mehrmaligem Aufrufen von oben nach unten aufeinanderfolgende Zeilen zurück.

```
>>> with open('file.txt') as f:
...     f.readline()
...     f.readline()
...     f.readline()
...     f.readline()
...     f.readline()
...     f.readline()
...
'Eins\n'
'Zwei\n'
'Drei\n'
'4 5 6 7 8\n'
'Ende\n'
''
```

Die leeren Strings in den letzten 3 Zeilen der Ausgabe weisen darauf hin, dass das Ende der Datei erreicht wurde und es nichts mehr zu lesen gibt.

Um Zeilen aus einer Datei zu lesen, kann auch mit einer `for`-Schleife über die Datei iteriert werden:

```
>>> with open('file.txt') as f:
...     for line in f: # Für jede Zeile 'line' in Datei 'f': ...
...         print(line)
...
Eins

Zwei

Drei

4 5 6 7 8

Ende
# Da print ein Newline setzt und bei Lesen der Datei ein Newline gelesen
# wird Besitzt die Ausgabe immer zwei Newlines pro Textzeile
```

2.8.3 Lesen der gesamten Datei in eine Liste

Eine weitere Möglichkeit ist, `readlines` zu verwenden. Mit dieser Funktion wird die ganze Datei auf einmal gelesen und jede Zeile als String in einer Liste gespeichert.

```
>>> with open('file.txt') as f:
...     f.readlines()
...
['Eins\n', 'Zwei\n', 'Drei\n', '4 5 6 7 8\n', 'Ende\n']

>>> liste = []
>>> with open('file.txt') as f:
...     liste = f.readlines()
...
>>> print(liste)
['Eins\n', 'Zwei\n', 'Drei\n', '4 5 6 7 8\n', 'Ende\n']
```

2.8.4 Lesen der gesamten Datei in einen String

Eine weitere Möglichkeit ist, `read` zu verwenden. Mit dieser Funktion wird die ganze Datei auf einmal gelesen und in einem String gespeichert.

```
with open('file.txt', 'r') as f:
    data=f.read()
# data = "Eins\nZwei\nDrei\n4 5 6 7 8\n"
```

oder:

```
data = open('file.txt', 'r').read()
# data = "Eins\nZwei\nDrei\n4 5 6 7 8\n"
```

Damit die Zeilenumbrüche in Spaces umgewandelt werden, kann man die `replace` Methode verwenden:

```
with open('file.txt', 'r') as f:
    data=f.read().replace('\n', ' ')
# data = "Eins Zwei Drei 4 5 6 7 8 "
```

2.8.5 `tell` und `seek`

`tell`

Je nachdem in welchem Modus man eine Datei öffnet, zeigt der File Pointer auf den Anfang oder auf das Ende der Datei (siehe Tabelle oben).

Folgende Datei `file.txt`

```
Eins
Zwei
Hallö
```

wird mit einem Python Programm eingelesen:

```
>>> f = open('file.txt', 'r')
```

Die Funktion `tell` gibt die Position des File Pointers zurück.

```
>>> f.tell()
0
```

Der Rückgabewert `0` bedeutet, dass, der File Pointer auf das erste Byte der Datei zeigt.

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	E	i	n	s	\n	Z	w	e	i	\n
FP	↑									

Index	10	11	12	13	14	15	16	17
Zeichen	H	a	l	l	ö		\n	
FP								

Ruft man nach dem Lesen einer Zeile mit `readline` noch ein Mal die `tell` Funktion auf, so sieht man, dass der File Pointer nun auf den Index `5`, also das sechste Byte der Datei zeigt.

```
>>> f.readline()
'Eins\n'
>>> f.tell()
5
```

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	E	i	n	s	\n	Z	w	e	i	\n
FP						↑				

Index	10	11	12	13	14	15	16	17
Zeichen	H	a	l	l	ö		\n	
FP								

Nach dem Lesen der nächsten Zeile zeigt der File Pointer auf das elfte Byte (Index 10).

```
>>> f.readline()
'Zwei\n'
>>> f.tell()
10
```

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	E	i	n	s	\n	Z	w	e	i	\n
FP										

Index	10	11	12	13	14	15	16	17
Zeichen	H	a	l	l	ö		\n	
FP	↑							

Nach dem Lesen der letzten Zeile zeigt der File Pointer auf Index 17.

```
>>> f.readline()
'Hallö\n'
>>> f.tell()
17
```

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	E	i	n	s	\n	Z	w	e	i	\n
FP										

Index	10	11	12	13	14	15	16	17
Zeichen	H	a	l	l	ö		\n	
FP								↑

In dieser letzten Zeile (`Hallö`) ist zu beachten, dass der Buchstabe `ö` zwei Bytes in Anspruch nimmt.

Ein weiterer Versuch weiterzulesen gibt einen leeren String zurück.

```
>>> f.readline()
```

```
''
>>> f.tell()
17
```

Das Ende der Datei ist erreicht.

`seek`

Um den File Pointer zu verschieben, kann die `seek`-Methode verwendet werden.

Der `seek`-Methode können zwei Argumente übergeben werden. Das erste bestimmt den Offset, und das zweite (optionale) relativ wozu dieser Offset ist.

Beginn	Erklärung
0	Anfang der Datei
1	
2	

2.8.6 Schreiben in eine Datei

Damit in eine Datei geschrieben werden kann, muss sie zum Schreiben geöffnet werden und dann können Strings in der Datei mit der Routine `write` gespeichert werden. Damit die Datei ordnungsgemäß abgeschlossen ist, muss nach der letzen Schreiboperation die Datei immer mit der Routine `close` geschlossen werden.

```
>>> f = open('fileout.txt', 'w')
>>> # nun kann in die Datei geschrieben werden
...
>>> f.close()
```

Achtung: In einer Datei können nur *strings* gespeichert werden. Sollen Zahlen und Objekte eines anderen Zahlentyps gespeichert werden, müssen sie zuerst in Strings konvertiert werden. Dazu hilft die Funktion `str`

```
>>> f = open('fileout.txt', 'w')
>>> f.write("Hier ist die Datei fileout.txt \n")
>>> i=100
>>> j=200
>>> f.write("Der Wert der Variable i = " + str(i))
>>> f.write("und der Wert der Variable j = " + str(j) + "\n")
>>> f.close()
```

Möchte man sich das `close` der Datei sparen, dann kann man wie beim Lesen mit dem

`with` statement arbeiten. Am Ende des `with` statements wird die Datei automatisch geschlossen.

```
>>> with open('fileout.txt', 'w') as file:
>>>     file.write('Hallo Zeile eins \n')
```

2.9 Interne Repräsentation von Zahlen

Das Dezimalsystem

Wenn wir eine Zahl, z.B. 253, lesen, gehen wir zunächst automatisch davon aus, dass es sich um eine Zahl des Dezimalsystems handelt. Im Zehnersystem wird mit 10 verschiedenen Ziffern gearbeitet, die durch die Zahlen 0 - 9 ausgedrückt werden.

Wir lesen die Ziffern nach folgendem Schema:
* Einer: 10^0
* Zehner: 10^1
* Hunderter: 10^2
* usw.

Daraus ergibt sich für die Zahl 253 folgende Darstellung(Polynom Darstellung):
$2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 = 253$

Analog geht man auch bei anderen Zahlensystemen vor, nur benutzt man andere Werte als Basis:

Das Binärsystem

Im Gegensatz zum Dezimalsystem, bei dem die Basis 10 verwendet wird, nimmt man beim Binärsystem die Basis 2 (= Übersetzung in Binärcode, wie er in Computern verwendet wird). Gültige Ziffern sind die 0 und die 1. Sollen nur positive Zahlen dargestellt werden, so kann man mit n Stellen Zahlen von 0 bis 2^{n-1} darstellen. Der darstellbare Zahlenbereich hat also die Größe 2^n.

Darstellung zu unterschiedlichen Basen:

Beispiel: Binärdarstellung einer Dezimalzahl

Dezimalzahl 11 = Binärzahl 1011

$1 \times 2^0 = 1$
$1 \times 2^1 = 2$
$0 \times 2^2 = 0$
$1 \times 2^3 = 8$
Summe = 11

Darstellung Negativer Werte:

a) Vorzeichenbit: das erste Bit zeigt das Vorzeichen an: 1 ist negativ, 0 ist positiv.

b) Zweierkomplement:

$b^t - x = 1 + \sum_{i=0}^{t-1} (b-1) \times b^i - \sum_{i=0}^{t-1} a_i \times b^i = 1 + \sum_{i=0}^{t-1} (b-1-a_i) \times b^i$

Darstellung im Zweierkomplement der Zahlen -4 bis +3

$-4 = 100$
$-3 = 101$
$-2 = 110$
$-1 = 111$

$0 = 000$
$1 = 001$
$2 = 010$
$3 = 011$

Darstellung reeller Werte:

$x = m \times b^e, e \in \mathbb{N}$

m ist die Mantisse, e der Exponent. Die Mantisse ist die Ziffernstelle einer Gleitkommazahl.

Beispiel: Bei der Zahl $5,5 \times 10^3$ ist $5,5$ die Mantisse, 3 der Exponent.

Das Oktalsystem

In diesem System ist die Basis 8, es sind Ziffern 0,1,2,3,4,5,6,7 gültig.

Beispiel: Oktalsystem

Die Dezimalzahl 136 wird dargestellt als
$2 \times 8^2 + 1 \times 8^1 + 0 \times 8^0 = 210$ oktal
53 (dezimal) = 65 (oktal) $= 6 \times 8^1 + 5 \times 8^0$
53 (dezimal) = 77 (oktal) $= 7 \times 8^1 + 7 \times 8^0$
136(dezimal) = 210(oktal) $= 2 \times 8^2 + 1 \times 8^1 + 0 \times 8^0$

Das Hexadezimalsystem

Beim Hexadezimalsystem wird die Zahl 16 as Basis zu Grunde gelegt. Für dieses System benötgt man 16 Ziffern, es werden zu den Ziffern 0-9 für die fehlenden 6 Ziffern die Buchstaben A-F verwendet.

Gültige “Ziffern”: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
Die Dezimalzahl 136 z.B. wird dargestellt als
$8 \times 16^1 + 8 \times 16^0 = 88$ hexadezimal

Das Hexadezimalsystem wird häufig verwendet, da es sich letztlich um eine komfortable Schrebweise des Binärsystems handelt.
Es eignet sich durch seine Basis der vierten Zweierpotenz ($16 = 2 \times 2 \times 2 \times 2$) zur relativ einfachen Datstellung von Binärzahlen. Ein Byte mit 8 Bit lässt sich in zwei Halbbyte mit 4 Bit aufteilen. Das heißt, man kann ein Byte mit zwei Stellen im Hxadezimalsystem darstellen.

Beispiel: Hexadezimalzahlen

53(dezimal) $= 35$(hexadezimal)
63(dezimal) $= 3F$(hexadezimal)

Beispiel: Darstellung der Dezimalzahl 60 im Speicher

$60/2 = 30$ Rest 0 (Wertigkeit 0. Stelle)
$30/2 = 15$ Rest 0 (Wertigkeit 1. Stelle)
$15/2 = 7$ Rest 1 (Wertigkeit 2. Stelle)
$7/2 = 3$ Rest 1 (Wertigkeit 3. Stelle)
$3/2 = 1$ Rest 1 (Wertigkeit 4. Stelle)
$1/2 = 0$ Rest 1 (Wertigkeit 5. Stelle)
das entspricht: | 1 | 1 | 1 | 1 | 0 | 0 |

Beispiel: Darstellung der Zahl 60 als Oktalzahl im Speicher

$60/8 = 7$ Rest4 (Wertigkeit 0. Stelle)
$7/8 = 0$ Rest7 (Wertigkeit 1. Stelle)
das entspricht: | 7 | 4 |

Beispiel: Zahlensysteme

12(dezimal)	1100 (binär)	14(oktal)	C(hexadezimal)
256(dezimal)	100000000 (binär)	400(oktal)	100(hexadezimal)
64(dezimal)	1000000 (binär)	100(oktal)	40(hexadezimal)

2.10 Interne Repräsentation von Schriftzeichen

2.10.1 Speicherung von Strings

Unter Strings werden in Python Folgen von Buchstaben verstanden.
Die Buchstaben, die den String repräsentieren, müssen in einfachen oder doppelten Anführungszeichen markiert werden. Im Speicher werden die Buchstaben entsprechend der fest vorgegebenen UNICODE-Kodierung als Zahlenwerte einzeln der Reihe nach abgespeichert. Jeder String kennt die Anzahl seiner Buchstaben.

2.10.2 Speicherung von Buchstaben

Da Buchstaben nicht direkt im Speicher abgebildet werden (können), wird jedem Zeichen eine Zahl (Zeichencode) zugeordnet. Die Abbildung "Buchstabe → Zahl" wurde im *American National Standard Code for Information Interchange* (ASCII) definiert, wobei dieser Code in den ersten Versionen nur 7 Bit lange Zahlen verwendete. Somit konnte diese Codierung nur 128 Zeichen kodieren, siehe Tabelle:

Tabelle 2.23: ASCII-Zeichentabelle

000	NUL	022	SYN	044	,	066	B	088	X	110	n
001	SOH	023	ETB	045	-	067	C	089	Y	111	o
002	STX	024	CAN	046	.	068	D	090	Z	112	p
003	ETX	025	EM	047	/	069	E	091	[	113	q
004	EOT	026	SUB	048	0	070	F	092	\	114	r
005	ENQ	027	ESC	049	1	071	G	093	]	115	s
006	ACK	028	FS	050	2	072	H	094	^	116	t
007	BEL	029	GS	051	3	073	I	095	_	117	u
008	BS	030	RS	052	4	074	J	096	‘	118	v
009	HT	031	US	053	5	075	K	097	a	119	w
010	LF	032		054	6	076	L	098	b	120	x
011	VT	033	!	055	7	077	M	099	c	121	y
012	FF	034	”	056	8	078	N	100	d	122	z
013	CR	035	#	057	9	079	O	101	e	123	{
014	SO	036	$	058	:	080	P	102	f	124	|
015	SI	037	#	059	;	081	Q	103	g	125	}
016	DLE	038	&	060	<	082	R	104	h	126	~
017	DC1	039	’	061	=	083	S	105	i	127	•
018	DC2	040	(	062	>	084	T	106	j		
019	DC3	041	)	063	?	085	U	107	k		
020	DC4	042	*	064	@	086	V	108	l		
021	NAK	043	+	065	A	087	W	109	m		

2.10.3 Bedeutung der Steuerzeichen

Die Buchstaben mit ASCII Code kleiner als 31 werden Steuerzeichen genannt und zur Formatierung von Texten oder Ausgaben verwendet.

Beispiel:

> Es ist folgender String gegeben:
> "abcd", Stringlänge ist 4.

Liegt eine ASCII-Kodierung vor, dann wird die Zeichenkette als Folge von 4 ganzen Zahlen (7-Bit Zahlen), entsprechend der ACSII-Codierung abgespeichert: 97,98,99,100 (Darstellung als Dezimalzahl)

Tabelle 2.24: Bedeutung der Steuerzeichen

	Layout Characters:		Escape Characters:
bs	backspace	so	shift-out
ht	horizontal tabulation	si	shift-in
lf	linefeed	esc	escape
vt	vertical tabulation		
ff	form feed		Medium Characters:
cr	carriage return	bel	bell ring
		dcX	device control (X: 1-4)
	Ignore Characters	em	end of medium
nul	null character		
can	cancel		Communication Characters:
sub	substitute	soh	start of heading
del	delete	stx	start of text
		etx	end of text
	Seperator Characters	eot	end of transmission
fs	file seperator	enq	enquiry
gs	group seperator	ack	acknowledgement
rs	record seperator	nak	negative acknowledgement
us	unit seperator	dle	data link escape
		syn	synchronous idle
		etb	end of transmission block

2.10.4 Zeichenkodierung: ISO-Latin-1 für Buchstaben, 8 Bit

Für länderspezifische Buchstaben fand sich im ASCII-Code kein Platz mehr. Versuche, den ASCII-Code auf 8 Bit zu erweitern, scheiterten daran, dass kein gemeinsamer Standard für die Buchstabennummern gefunden wurde. Erst der ISO (International Standardization Organization) gelang es den Buchstabencode auf 8 Bit zu erweitern und länderspezifische Buchstaben zu standardisieren. Die ISO definierte für verschieden Sprachfamilien verschiedene Zeichenkodierungen. Für die lateinischen Buchstaben definierten sie z.B. ISO-LATIN(ISO-8859-1). Um Buchstaben zu kodieren, die nicht lateinisch sind, wurden von der ISO weitere länderspezifische Zeichencodes entwickelt, z.B. für die griechischen Buchstaben ISO-Greek(ISO-8859-7).

Tabelle 2.25: ISO-Latin-1 Kodierung

Zeichen	Beschreibung	HTML-Name	Hex.	Dezimal
	erzwungenes Leerzeichen	` `	` `	160
¡	umgekehrtes Ausrufezeichen	`&iexcl;`	`&#xA1;`	161
¢	Cent-Zeichen	`&cent;`	`&#xA2;`	162
£	Pfund-Zeichen	`&pound;`	`&#xA3;`	163
¤	Währungszeichen	`&curren;`	`&#xA4;`	164
¥	Yen-Zeichen	`&yen;`	`&#xA5;`	165
¦	durchbrochener Strich	`&brvbar;`	`&#xA6;`	166
§	Paragraph-Zeichen	`&sect;`	`&#xA7;`	167
¨	Pünktchen oben (für Umlaut)	`&uml;`	`&#xA8;`	168
©	Copyright-Zeichen	`&copy;`	`&#xA9;`	169
ª	Ordinal-Zeichen weiblich	`&ordf;`	`&#xAA;`	170
«	angewinkelte Anführungszeichen links	`&laquo;`	`&#xAB;`	171
¬	Verneinungs-Zeichen	`&not;`	`&#xAC;`	172
	bedingter Trennstrich	`&shy;`	`&#xAD;`	173
®	Registriermarke-Zeichen	`&reg;`	`&#xAE;`	174
¯	Überstrich (Macron)	`&macr;`	`&#xAF;`	175
°	Grad-Zeichen	`&deg;`	`&#xB0;`	176
±	Plusminus-Zeichen	`&plusmn;`	`&#xB1;`	177
²	Hoch-2-Zeichen	`&sup2;`	`&#xB2;`	178
³	Hoch-3-Zeichen	`&sup3;`	`&#xB3;`	179
´	Akut-Zeichen	`&acute;`	`&#xB4;`	180
µ	Mikro-Zeichen	`&micro;`	`&#xB5;`	181
¶	Absatz-Zeichen	`&para;`	`&#xB6;`	182
·	Mittelpunkt	`&middot;`	`&#xB7;`	183
¸	Häkchen unten	`&cedil;`	`&#xB8;`	184
¹	Hoch-1-Zeichen	`&sup1;`	`&#xB9;`	185
º	Ordinal-Zeichen männlich	`&ordm;`	`&#xBA;`	186
»	angewinkelte Anführungszeichen rechts	`&raquo;`	`&#xBB;`	187
¼	ein Viertel	`&frac14;`	`&#xBC;`	188

Zeichen	Beschreibung	HTML-Name	Hex.	Dezimal
½	ein Halb	`&frac12;`	`&#xBD;`	189
¾	drei Viertel	`&frac34;`	`&#xBE;`	190
¿	umgekehrtes Fragezeichen	`&iquest;`	`&#xBF;`	191
À	A mit accent grave (Gravis)	`&Agrave;`	`&#xC0;`	192
Á	A mit accent aigu (Akut)	`&Aacute;`	`&#xC1;`	193
Â	A mit Zirkumflex	`&Acirc;`	`&#xC2;`	194
Ã	A mit Tilde	`&Atilde;`	`&#xC3;`	195
Ä	A Umlaut	`&Auml;`	`&#xC4;`	196
Å	A mit Ring	`&Aring;`	`&#xC5;`	197
Æ	AE-Ligatur	`&AElig;`	`&#xC6;`	198
Ç	C mit Häkchen (Cedille)	`&Ccedil;`	`&#xC7;`	199
È	E mit accent grave (Gravis)	`&Egrave;`	`&#xC8;`	100
É	E mit accent aigu (Akut)	`&Eacute;`	`&#xC9;`	101
Ê	E mit Zirkumflex	`&Ecirc;`	`&#xCA;`	102
Ë	E Umlaut	`&Euml;`	`&#xCB;`	103
Ì	I mit accent grave (Gravis)	`&Igrave;`	`&#xCC;`	104
Í	I mit accent aigu (Akut)	`&Iacute;`	`&#xCD;`	105
Î	I mit Zirkumflex	`&Icirc;`	`&#xCE;`	106
Ï	I Umlaut	`&Iuml;`	`&#xCF;`	107
Đ	großes Eth (isländisch)	`&ETH;`	`&#xD0;`	108
Ñ	N mit Tilde	`&Ntilde;`	`&#xD1;`	109
Ò	O mit accent grave (Gravis)	`&Ograve;`	`&#xD2;`	110
Ó	O mit accent aigu (Akut)	`&Oacute;`	`&#xD3;`	111
Ô	O mit Zirkumflex	`&Ocirc;`	`&#xD4;`	112
Õ	O mit Tilde	`&Otilde;`	`&#xD5;`	113
Ö	O Umlaut	`&Ouml;`	`&#xD6;`	114
×	Mal-Zeichen	`&times;`	`&#xD7;`	115
Ø	O mit Schrägstrich	`&Oslash;`	`&#xD8;`	116
Ù	U mit accent grave (Gravis)	`&Ugrave;`	`&#xD9;`	117
Ú	U mit accent aigu (Akut)	`&Uacute;`	`&#xDA;`	118
Û	U mit Zirkumflex	`&Ucirc;`	`&#xDB;`	119
Ü	U Umlaut	`&Uuml;`	`&#xDC;`	120
Ý	Y mit accent aigu (Akut)	`&Yacute;`	`&#xDD;`	121
Þ	großes Thorn (isländisch)	`&THORN;`	`&#xDE;`	122
ß	scharfes S (sz-Ligatur)	`&szlig;`	`&#xDF;`	123
à	a mit accent grave (Gravis)	`&agrave;`	`&#xE0;`	124
á	a mit accent aigu (Akut)	`&aacute;`	`&#xE1;`	125
â	a mit Zirkumflex	`&acirc;`	`&#xE2;`	126
ã	a mit Tilde	`&atilde;`	`&#xE3;`	127
ä	a Umlaut	`&auml;`	`&#xE4;`	128
å	a mit Ring	`&aring;`	`&#xE5;`	129
æ	ae-Ligatur	`&aelig;`	`&#xE6;`	130

Zeichen	Beschreibung	HTML-Name	Hex.	Dezimal
ç	c mit Häkchen (Cedille)	`&ccedil;`	`&#xE7;`	131
è	e mit accent grave (Gravis)	`&egrave;`	`&#xE8;`	132
é	e mit accent aigu (Akut)	`&eacute;`	`&#xE9;`	133
ê	e mit Zirkumflex	`&ecirc;`	`&#xEA;`	134
ë	e Umlaut	`&euml;`	`&#xEB;`	135
ì	i mit accent grave (Gravis)	`&igrave;`	`&#xEC;`	136
í	i mit accent aigu (Akut)	`&iacute;`	`&#xED;`	137
î	i mit Zirkumflex	`&icirc;`	`&#xEE;`	138
ï	i Umlaut	`&iuml;`	`&#xEF;`	139
ð	kleines Eth (isländisch)	`&eth;`	`&#xF0;`	140
ñ	n mit Tilde	`&ntilde;`	`&#xF1;`	141
ò	o mit accent grave (Gravis)	`&ograve;`	`&#xF2;`	142
ó	o mit accent aigu (Akut)	`&oacute;`	`&#xF3;`	143
ô	o mit Zirkumflex	`&ocirc;`	`&#xF4;`	144
õ	o mit Tilde	`&otilde;`	`&#xF5;`	145
ö	o Umlaut	`&ouml;`	`&#xF6;`	146
÷	Divisions-Zeichen	`&divide;`	`&#xF7;`	147
ø	o mit Schrägstrich	`&oslash;`	`&#xF8;`	148
ù	u mit accent grave (Gravis)	`&ugrave;`	`&#xF9;`	149
ú	u mit accent aigu (Akut)	`&uacute;`	`&#xFA;`	150
û	u mit Zirkumflex	`&ucirc;`	`&#xFB;`	151
ü	u Umlaut	`&uuml;`	`&#xFC;`	152
ý	y mit accent aigu (Akut)	`&yacute;`	`&#xFD;`	153
þ	kleines Thorn (isländisch)	`&thorn;`	`&#xFE;`	154
ÿ	y Umlaut	`&yuml;`	`&#xFF;`	155

2.10.5 Zeichenkodierung nach den Vorgaben des UNICODE Konsortiums

Der große Nachteil der ISO-Standardisierung ist, dass ähnlich wie bei ASCII nur einem Teil der sinntragenden Schriftzeichen bzw. Textelemente aller bekannten Schriftkulturen und Zeichensysteme ein Zahlencode zugeordnet wurde. Außerdem gab es bei den Zeichencodes keine Eindeutigkeit, da die Zuordnung des Zahlencodes zum Schriftzeichen von der eingestellten Sprachfamilie abhängig war:

Zahlencode (Hex)	ISO-Greek	ISO-Latin
C4	Δ (Delta)	Ä (Umlaut Ä)

Die Ambiguität löste das UNICODE Konsortium, indem es ein Universal Character Set (UCS) definierte. Der UCS reserviert für 1.114.112 Schriftzeichen Zeichencodes. Sie sollen für alle Schriftzeichen aller bekannten Schriftkulturen ausreichen. Die Bitbreite für die

Zeichencodes wurde von 8 Bit auf 32 Bit erweitert. Das UNICODE Konsortium definierte eine Tabelle, in der jedem Schriftzeichen ein eindeutiger Zeichencode zugeordnet wurde:

Tabelle 2.27: UNICODE-Codierung

Zahlencode (Hex)	Zeichen
41	A
42	B
C4	Ä
3FF	Δ (griechisches Delta)
22FF	∆ (mathematisches Delta)
C93E	Þ (Großbuchstabe “Thorn”)

Würde jetzt Text mit den vier Zechen ABBA abgespeichert, so wäre dies unter UNICODE mit der Folge von vier 32-Bit-Zahlen realisiert:

Bit 1	Bit 2	Bit 3	Bit 4
0	0	0	41
0	0	0	42
0	0	0	42
0	0	0	41

Wir hätten also 4×4 Bytes, also 16 Bytes in der Datei. Betrachtet man die 16 Bytes der Zeichencodes, so sieht man, dass nur 4 Bytes Informationen tragen, die anderen 12 Bytes sind Nullbytes. Als C-Programmierer weiß man, welche Probleme die Zahl Null in einem bytestream verursachen kann (Stichwort: File/ Stringende). Außerdem kann man auch von einer Platzverschwendung sprechen, da 16 Bytes belegt sind und nur 4 Bytes die Information der Zeichencodes tragen. Daraufhin entwickelte Ken Thompson von den BELL Laboratories ein Transformationsformat, das in eindeutiger Weise die Zahlenfolge repräsentiert, aber nur die Bytes speichert, die die Informationen für die Zeichencodes tragen. Auf die Nullbytes verzichtet sein Verfahren. Das UNICODE – Konsortium übernahm dieses Verfahren und legte für jedes Schriftzeichen eine eindeutige Folge von Zahlen fest. Ein mitgelieferter Transformationsalgorithmus kann aus dieser Folge die eindeutige UNICODE-Zahl des zugrunde gelegten Schriftzeichens berechnen. Diese Kodierung nannten Sie auch „multibyte-Kodierung“. Aus „ABBA“ wurde wieder die Folge der 4 Bytes: 41 42 42 41

2.10.6 Die UTF-Prinzipien

Das UNICODE-Transformation-Format (UTF) weist jedem Schriftzeichen eien eindeutige Folge von Zahlen zu, deren Bitbreite das Appedndix “-n” ausdrückt:

UTF-8 … eine Folge von 8-Bit Zahlen (1 Byte)
UTF-16 … eine Folge von 16-Bit Zahlen (2 Bytes)
UTF-32 … eine Folge von 32-Bit Zahlen (4 Bytes)

Kennt der Transformations Algorithmus die Bitbreite der zugrundegelegten Codierung, dann kann er die eindeutige UNICODE-Nummer mit Hilfe von Bitoperationen sehr effizient berechnen.

Schriftzeichen	UNICODE Nummer (Hexadezimal)	UTF-8 Bitfolge (Hexadezimal)
A	41	41
Ä	C4	C3 84
Δ	394	CE 94

Der Text aus den drei Zeichen A, Ä und Delta Δ: “AÄΔ” wäre bei UTF-8 mit folgender Bytefolge abgespeichert:

41 C3 84 CE 94

Wie der Transformations-Algorithmus aus dieser Folge die Zeichennummer 41, C4 und 394 errechnet, wird im nächsten Kapitel erklärt.

2.10.7 Die UTF Kodierung im Detail

aus http://de.wikipedia.org/wiki/UTF-8:
“UTF-8 ist die fortschrittlichste und populärste Kodierung für Unicode-Zeichen; dabei wird jedem Unicode-Zeichen eine speziell kodierte Bytekette von variabler Länge zugeordnet. Es unterstützt bis zu 4 Byte auf die sich wie bei allen UTF Formaten alle 1.114.112 Unicode Zeichen abbilden lassen. UTF-8 ist gegenwärtig als Request for Comments RFC 3629 standardisiert (UTF-8, a transformation format of ISO 10646). Dieser Standard löst das ältere RFC 2279 ab. Unicode-Zeichen mit den Werten aus dem Bereich von 0 bis 127 (0 bis 7F hexadezimal) werden in der UTF-8-Kodierung als ein Byte mit dem gleichen Wert wiedergegeben. Insofern sind alle Daten, die ausschließlich echte ASCII-Zeichen verwenden, in beiden Darstellungen identisch. Unicode-Zeichen größer als 127 werden in der UTF-8-Kodierung zu Byteketten der Länge zwei bis vier.”

Tabelle: UTF-Kodierung im Detail

Unicode-Bereich	UTF-8 Kodierung	Bemerkungen	Möglichkeiten	
0000 0000 - 0000 007F	0xxxxxxx	In diesem Bereich (128 Zeichen) entspricht UTF-8 genau dem ASCII-Code: Das erste Bit ist *0*, die darauf folgende 7-Bitkombination ist das ASCII-Zeichen.	2^7	128
0000 0080 - 0000 07FF	110xxxxx 10xxxxxx	Das erste Byte beginnt mit binär 11, die folgenden Bytes beginnen mit binär 10; die x stehen für die fortlaufende Bitkombination des Unicodezeichens. Die Anzahl der Einsen bis zur ersten 0 im ersten Byte	$2^{11} - 2^7$ [2^{11}]	1.920 [2.048]
0000 0800 - 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx		$2^{16} - 2^{11}$ [2^{16}]	63.488 [65.536]
0001 0000 - 0010 FFFF [0001 0000 - 001F FFFF]	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	ist die Anzahl der Bytes für das Zeichen. [In eckigen Klammern jeweils die theoretisch maximal möglichen.]	2^{20} [2^{21}]	1.048.576 [2.097.152]

Der Algorithmus lässt theoretisch mehrere Billionen Zeichen zu.
$(2^{7\times6} + 2^{6\times6+1} + 2^{5\times6+2} + \cdots)$
Die aktuelle RFC beschränkt diese jedoch auf die durch UTF-16 erreichbaren, also bis 0010 FFFF (1.114.111). Das erste Byte eines UTF-8-kodierten Zeichens nennt man dabei Start-Byte, weitere Bytes nennt man Folgebytes. Startbytes beginnen mit der Bitfolge 11 oder einem 0-Bit, während Folgebytes immer mit der Bitfolge 10 beginnen. Betrachtet man die Bitfolgen etwas genauer, erkennt man die große Sinnfälligkeit von UTF-8:
Ist das höchste Bit des ersten Byte 0, handelt es sich um ein gewöhnliches ASCII-Zeichen, da ASCII eine 7-Bit-Kodierung ist und die ersten 128 Zeichen des Unicode die ASCII-Zeichen sind. Damit sind alle ASCII-Dokumente automatisch aufwärtskompatibel zu UTF-8.
Ist das höchste Bit des ersten Byte 1, handelt es sich um ein Mehrbytezeichen, also ein Unicode-Zeichen mit einer Zeichennummer größer als 127. Sind die höchsten beiden Bits des ersten Byte 11, handelt es sich um das Start-Byte eines Mehrbytezeichens, sind sie 10, um ein Folge-Byte.

Die lexikalische Ordnung nach Byte-Werten entspricht der lexikalischen Ordnung nach Buchstaben-Nummern, da größere Zeichennummern mit entsprechend mehr 1-Bits am

Anfang kodiert werden.

Bei den Start-Bytes von Mehrbyte-Zeichen gibt die Anzahl der 1-Bits vor dem ersten 0-Bit die gesamte Bytezahl des als Mehrbyte-Zeichen kodierten Unicode-Zeichens an. Anders interpretiert, die Anzahl der 1-Bits vor dem ersten 0-Bit nach dem ersten Bit entspricht der Anzahl an Folgebytes, z.B. 1110xxxx 10xxxxxx 10xxxxxx:
3 Bits vor dem ersten 0-Bit = 3 Bytes insgesamt, 2 Bits vor dem ersten 0-Bit nach dem ersten 1-Bit = 2 Folgebytes.

Start-Bytes (0xxx xxxx oder 11xx xxxx) und Folge-Bytes (10xx xxxx) lassen sich eindeutig voneinander unterscheiden. Somit kann ein Byte-Strom auch in der Mitte gelesen werden, ohne dass es Probleme mit der Dekodierung gibt, was insbesondere bei der Wiederherstellung defekter Daten wichtig ist. Bytes, die mit 10 beginnen, werden einfach übersprungen, bis ein Byte gefunden wird, das mit 0 oder 11 beginnt. Könnten Start-Bytes und Folge-Bytes nicht eindeutig voneinander unterschieden werden, wäre das Lesen eines UTF-8-Datenstroms, dessen Beginn unbekannt ist, unter Umständen nicht möglich.

Zu beachten:

Das gleiche Zeichen kann theoretisch auf verschiedene Weise kodiert werden. Jedoch ist nur die jeweils kürzeste mögliche Kodierung erlaubt. Bei mehreren Bytes für ein Zeichen werden die Bits rechtsbündig angeordnet - das rechte Bit des Unicode Zeichens steht also immer im rechten Bit des letzten UTF-8 Bytes. Ursprünglich gab es auch Kodierungen mit mehr als 4 Oktetts (bis zu 6), diese sind jedoch ausgeschlossen worden, da es in Unicode keine korrespondierenden Zeichen gibt und ISO 10646 in seinem möglichen Zeichenumfang an Unicode angeglichen wurde. Für alle auf dem Lateinischen Alphabet basierten Schriften ist UTF-8 die platzsparendste Methode zur Abbildung von Unicode Zeichen. UTF-8, UTF-16 und UTF-32 kodieren alle den vollen Wertebereich von Unicode.

2.10.8 UTF-8 im Internet

Im Internet wird immer häufiger die UTF-8 Kodierung verwendet. So unterstützt auch Wikipedia alle denkbaren Sprachen. Möglich macht all das ein Meta-Tag im Kopf der Website:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Praktisch jeder Browser unterstützt heutzutage die Unicode-Formate.

Auch in e-Mails ist bei vielen Programmen schon die UTF-8 Kodierung voreingestellt. Sie stellt sicher, dass auch Sonderzeichen unterschiedlicher Länder richtig übertragen und dargestellt werden.

2.10.9 Unicode in Python

Seit Python 3 wird Unicode in Python nativ unterstützt. Noch in Python 2 war ASCII der Standard! Dass Python 3 Unicode nativ unterstützt bedeutet für den Programmierer, dass er ohne großen Aufwand

- Unicode-Dateien einlesen kann
- Unicode-Dateien generieren und ausgeben kann
- Unicode-Zeichen im Quelltext seines Programmes verwenden kann

Unicode im Programmcode

Der Python-Interpreter unterstützt den Unicode-Standard. Es können deswegen beliebige Unicode-Zeichen im Quelltext des Programmes als Literale angegeben werden, sofern die .py-Datei auf dem Rechner anschließend als eine Unicode-Datei gespeichert wird. Das hängt von den Einstellungen des Texteditors ab und wird von jedem modernen Editor unterstützt. Auf Unix-basierten Systemen ist UTF-8 sogar der Standard des Betriebssystems.

```
>>> список = [1,2,3,4,5]
>>> print(список)
[1, 2, 3, 4, 5]
>>> giriş = "自由"
>>> print(giriş)
'自由'
```

Wenn die Tastatur oder der Editor ein Zeichen nicht unterstützen, so kann auch der Unicode-Zahlenwert des Zeichens direkt angegeben werden. Das Zeichen wird dann vom Interpreter als normales Unicode-Zeichen verwendet:

```
>>> my_unicode_string = "\u1111\u1112\u1110\u1104"
>>> print(my_unicode_string)
'太空旅行'
```

Ein 16-Bit-Unicode-Zeichen wird innerhalb eines Strings durch einen Backslash (`\`), gefolgt von einem kleinen `u`, gefolgt vom hexadezimalen Zahlenwert des Unicode-Zeichens angegeben: `\u1A1F`

Weil Python 3 mit dem Datentypten String (`str`) Unicode vollständig unterstützt, spielt die tatsächlich kodierte Byte-Länge eines Zeichens (z.B. in UTF-8) keine Rolle. Ein Buchstabe wird in einem String immer die Länge 1 haben, auch wenn das Zeichen einen hohen Unicode-Zahlenwert hat, und damit im Speicher mehr als ein Byte belegt.
Ein String-Objekt aus 4 lateinischen Buchstaben hat demzufolge ebenso die Länge 4, wie ein String aus 4 chinesischen Schriftzeichen, auch wenn sich die Länge der Bytefolgen auf der Festplatte sehr wahrscheinlich unterscheidet.

Kodierungen beim Lesen und Schreiben von Dateien

Beim Öffnen von Dateien mit dem `open()`-Statement kann optional angegeben werden, mit welchem Encoding die Datei gelesen oder geschrieben werden soll. Wenn vom Programmierer nichts weiter angegeben wird, so wird das auf dem System als Standard geltende Encoding verwendet.
Möchte man herausfinden, welches Encoding dies auf dem aktuellen Rechner ist, so funktioniert das über eine Funktion aus dem Modul `locale`:

```
>>> import locale
>>> locale.getpreferredencoding(False)
'UTF-8'
```

Dieses Verhalten ist der Grund, warum es mit Python 3 in der Regel problemlos möglich ist, ASCII-, ISO-Latin-1-, und UTF-Dateien einzulesen: ASCII und ISO-Latin-1 sind deckungsgleich mit einem Teil der Unicode-Zeichentabelle.

```
>>> file = open("my_utf8_file.txt", "r")
>>> file.readline()
'□ □ □ □'
```

Bislang wurden beim Öffnen von Dateien mit dem `open()`-Statement zwei Argumente betrachtet: `open("file.txt", mode="r")`. Mit einem zusätzlichen benannten Argument lässt sich beim `open`-Statement das Encoding angeben:

```
>>> f = open("my_UTF8_file.txt", mode="r", encoding="UTF-8")
>>> g = open("my_isolatin_file.txt", mode="w", encoding="iso-8859-1")
>>> h = open("my_isolatin_file.txt", mode="w", encoding="Latin_1")
>>> i = open("my_ascii_file.txt", "a", encoding="ascii") # mode= ist optional, encoding= is
```

Beachte: Wegen der Reihenfolge der potentiellen Argumente von `open` ist es nötig, das Encoding (`encoding=`) als **benanntes Argument** anzugeben.

Sämtliche in Python 3 verfügbaren Encodings finden sich auf der Seite der Python-Dokumentation: https://docs.python.org/3/library/codecs.html#standard-encodings

Alle Encoding-Namen sind case-insensitive.

Zur Wiederholung: Lesen aus einer Datei

- Lesen der gesamten Datei text.txt: Zeilenweise in eine Liste:

```
>>> with open('text.txt',mode='r', encoding='UTF-8') as f:
>>>      lines=f.readlines()
>>> print(lines)
['Zeile eins \n', 'ÄÄ ÜÜ öö\n', '3\n', 'Tschüüßßß\n']
```

- Lesen einer Zeile aus einer Datei text.txt

```
>>> with open('text.txt',mode='r', encoding='UTF-8') as f:
>>>      line=f.readline()
>>> print(line)
Zeile eins
```

2.11 Weitere Datenstrukturen: Tupel und Dictionaries

2.11.1 Tupel

Listen sind in Python sehr mächtig und flexibel, benötigen dadurch aber auch vergleichsweise viel Rechenaufwand. Vor allem in laufzeitintensiven Anwendungen wird deshalb oft der Datentyp `tuple` eingesetzt, der die Funktionalitäten einschränkt, dafür aber viel Rechenleistung spart.

Tupel können also als ein "Sonderfall" von Listen betrachtet werden, was sich in ihrer Syntax widerspiegelt: Sie ist fast identisch zu der von Listen.

Tupel werden initialisiert, indem der Inhalt in runde Klammern gesetzt wird:

Beispiel: Initialisierung von Tupeln

```
zahlen = (1,2,3,4)
worte = ("Haus","Hof","Kind")
gemischtes = ("Merlin", 7, 2.7)
```

Auf die Elemente eines Tupels wird über die normale Listen-Syntax zugegriffen:

Beispiel: Zugriff auf Elemente eines Tupels

```
>>> vektor = (11.1, 45.7, 3.8)
>>> vektor[0]
11.1
>>> vektor[2]
3.8
```

Unterschiede eines Tupels von einer Liste:

- Die Länge eines Tupels kann sich nicht nachträglich verändern
- Der Inhalt eines Tupels kann sich nicht nachträglich verändern. Achtung: Ist der Inhalt eines Tupels eine Liste oder ein Dictionary, so kann sich *dessen* Inhalt weiterhin verändern

2.11.2 Dictionaries

```
wörterbuch = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
```

Neben der Möglichkeit Elemente in Listen zu speichern und über Indizes auf sie zuzugreifen gibt es in Python die Möglichkeit, Elemente in sogenannten Dictionaries abzuspeichern. Bei Dictionaries hat man die Möglichkeit den Schlüssel, mit dem man auf die Elemente zugreift, selbst festzulegen.

- Die Schlüssel, mit denen man bei Dictionaries auf die Elemente zugreift nennt man `key`

- Den Wert, der dem Schlüssel zugeordnet ist, nennt man `value`
- Dictionaries werden vom Datentypen `dict` dargestellt.

In obigem Beispiel wird also dem Schlüssel `"Apfel"` der Wert `"apple"` zugeordnet, dem Schlüssel `"Birne"` der Wert `"pear"` und so weiter.

- Die Schlüssel eines Lexikons müssen eindeutig sein. Formal ist es zwar möglich, einen Schlüssel mehrmals zu setzen, jedoch wird dann der vorherige Wert überschrieben.
- Als Schlüssel dürfen beliebige Datentypen, aber keine *mutable*-Datentypen (Liste, Dictionary) verwendet werden
- Die Werte eines Lexikons hingegen dürfen beliebig oft vorkommen, da sie eindeutig einem Schlüssel zugeordnet sind. Ganz wie bei Schließfächern: Es kann jede Schließfachnummer nur einmal geben, aber in jedem Schließfach kann ein Kaninchen sitzen.

Auf einen Wert im Dictionary kann man genauso zugreifen, wie auf ein Element einer Liste, nur, dass statt dem Index der Schlüssel angegeben wird:

Beispiel: Zugriff auf ein Dictionary

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> lexikon["Apfel"]
'apple'
>>> lexikon["Tomate"]
'tomato'
```

Genauso wird ein Schlüssel-Wert-Paar in einem bestehenden Dictionary gesetzt:

Beispiel: Zuweisung an ein Dictionary

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> lexikon["Apfel"] = "la mela"
>>> lexikon["Apfel"]
'la mela'
>>> lexikon["Lastkraftwagenfahrer"] = "truck driver"
>>> lexikon["Lastkraftwagenfahrer"]
'truck driver'
>>> print(lexikon)
{'Lastkraftwagenfahrer': 'truck driver', 'Tomate': 'tomato',
'Apfel': 'la mela', 'Birne': 'pear'}
```

Bislang wurden nur Strings als Schlüssel und Werte verwendet, doch in ein und demselben Dictionary können beliebige Datentypen verwendet und gemischt werden. Beachten Sie aber, dass die beiden *mutable*-Datentypen Liste und Dictionary nicht als Schlüssel verwendet werden können, da ein Schlüssel, der seine Beschaffenheit verändern kann, nicht eindeutig ist.

Beispiel: Gemischte Datentypen

```
>>> zuordnungen = {1: "Eins", "Alter": 82, \
"Inhalt": ["Schnittlauch","Mehl"], 3.14: True}
```

Das Python-Dictionary entspricht der Hash-Map in vielen anderen Programmiersprachen

2.11.3 Iteration über ein Dictionary

Über ein Dictionary kann mit der `for`-Schleife iteriert werden, ganz wie bei Tupel und Liste. Dann wird die Schleifenvariable nacheinander jeden Schlüssel des Dictionarys enthalten:

Beispiel: Iteration über die Schlüssel eines Dictionarys

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for key in lexikon:
...     print(key)
...
'Apfel'
'Birne'
'Tomate'
```

Dies ist äquivalent zu:

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for key in lexikon.keys():
...     print(key)
...
'Apfel'
'Birne'
'Tomate'
```

Möchte man auf diese Weise auf die Werte des Dictionarys zugreifen, so kann man die normale Zugriffssyntax verwenden:

Beispiel: Zugriff auf die Werte des iterierten Dictionaries

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for key in lexikon:
...     print(lexikon[key])
...
'apple'
'pear'
'tomato'
```

Es ist über die `.values()`-Funktion ebenfalls möglich direkt über die Werte zu iterieren:

Beispiel: Iteration über die Werte eines Dictionaries

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for val in lexikon.values():
...     print(val)
...
'apple'
'pear'
'tomato'
```

Und, der Vollständigkeit halber, ebenso über alle Schlüssel-Wert-Paare mittels der `.items`-Funktion. Die Schlüssel-Wert-Paare werden jeweils als ein Tupel ausgegeben:

Beispiel: Iteration über die Schlüssel-Wert-Paare eines Dictionarys

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for eintrag in lexikon.items():
...     print(eintrag)
...
('Apfel', 'apple')
('Tomate', 'tomato')
('Birne', 'pear')
```

Folglich wäre in obrigem Beispiel `eintrag[0]` jeweils der Key jedes Dictionary-Eintrags.

Zuletzt lässt uns Python diese Key-Value-Tupel auch direkt in Variablen "auspacken", indem wir der for-Schleife zwei Schleifenvariablen zuweisen:

Beispiel: Auto-Unpackaging der Tupel

```
>>> lexikon = {"Apfel": "apple", "Birne": "pear", "Tomate": "tomato"}
>>> for key, value in lexikon.items():
...     print(key, "->", value)
...
Apfel -> apple
Tomate -> tomato
Birne -> pear
```

3 Reguläre Ausdrücke

Zur Suche von Zeichenfolgen innerhalb von Strings haben wir bisher nur den Vergleichsoperator und den Zugriff auf die einzelnen Buchstaben kennen gelernt. In Python kann man aber mit Hilfe von regulären Ausdrücken beliebige Zeichenfolgen innerhalb eines Strings auffinden. Unter einem regulären Ausdruck verstehen wir eine Folge von Zeichen und Metazeichen, die eine Klasse von Zeichenfolgen beschreibt.

Ein regulärer Ausdruck ist ein spezieller Text, der ein Suchmuster beschreibt. Beispielsweise würde dieser Ausdruck

```
[^@]+@[^@]+\.[^@]+
```

alle Zeichenketten finden, die einen String enthalten, der wie eine E-Mail-Adresse aussieht. In diesem Fall werden alle Strings gefunden, in welchen

- genau ein `@` vorkommt
- vor dem `@` mindestens ein Zeichen ist (aber kein @)
- in dem Teil nach dem `@` genau ein Punkt ist
- vor und nach diesem Punkt mindestens ein Zeichen ist (aber kein @).

Mit `s[ae]p[ae]r[ae]te` kann man das Wort `separate` und häufige Fehlschreibungen finden.

Tabelle 3.1: Patterns in regulären Ausdrücken

Pattern	Beschreibung
`^`	Matched Anfang eines Strings.
`$`	Matched Ende eines Strings.
`.`	Matched jedes Zeichen außer newline.
`[...]`	Matched jedes Zeichen innerhalb der Klammern.
`[^...]`	Matched jedes Zeichen, welches nicht innerhalb der Klammern ist.
`re*`	Matched 0 oder mehr Vorkommen des vorangehenden Ausdrucks.
`re+`	Matched 1 oder mehr Vorkommen des vorangehenden Ausdrucks.
`re?`	Matched 0 oder 1 Vorkommen des vorangehenden Ausdrucks.
`re{n}`	Matched genau n Vorkommen des vorgangehenden Ausdrucks.
`re{n,}`	Matched n oder mehr Vorkommen des vorangehenden Ausdrucks.
`re{n,m}`	Matched mindestens n, höchstens m Vorkommen des vorangehenden Ausdrucks.
`a|b`	Matched `a` oder `b`.

Pattern	Beschreibung
`(re)`	Gruppiert Reguläre Ausdrücke und merkt sich gematchten Text.
`\w`	Matched Wort Zeichen.
`\W`	Matched Nicht-Wort Zeichen.
`\s`	Matched Whitespace. Äquivalent zu `[\t\n\r\f]`.
`\S`	Matched Nicht-Whitespace.
`\d`	Matched Ziffern. Äquivalent zu `[0-9]`.
`\D`	Matched Zeichen, die keine Ziffern sind.
`\A`	Matched Anfang eines Strings.
`\Z`	Matched Ende eines Strings. Existiert newline, dann matched es direkt davor.
`\z`	Matched Ende eines Strings.
`\G`	Matched den Punkt, wo der letzte Match zu Ende war.
`\b`	Matched Wortgrenzen, wenn außerhalb von Klammern. Matched backspace (`0x08`) wenn innerhalb von Klammern.
`\B`	Matched Zeichen, die keine Wortgrenzen sind.
`\n`, `\t`, etc.	Matched newlines, tabs, carriage returns, etc.

Tabelle 3.2: Beispiele für reguläre Ausdrücke

Beispiele	Erklärung
`^abc`	Matched z.B. `abcdef`, aber nicht `aaaabc`
`xyz$`	Matched z.B. `uvwxyz`, aber nicht `vwxyza`
`.reis.`	Matched z.B. `greise`, `abreise`, `abreisen`, aber nicht `greis`
`ventil.tor`	Matched z.B. `ventilator`, aber nicht `ventil\ntor`
`[aeiou]h`	Matched z.B. `ah`, `eh`, `ih`, `bah`, `buh`, `baha`, `buhu` aber nicht `aeiouh`
`[A-E]`	Matched z.B. `A`, `BD`, `DXY`, `Eabc`, aber nicht `a`, `ae`, oder `FFFF`
`[^a]`	Matched z.B. `pferd`, `horse`, `cheval`, `aaabaaa`, aber nicht `aaaaaaaa`
`a*`	Matched z.B. ϵ (leerer String), `a`, `b`, `aaaaaa`, `baaaaaa`, `aaaah`
`a+`	Matched z.B. `a`, `aaaaaa`, `baaaaaa`, `aaaah`, aber nicht ϵ (leerer String), `b`
`a?`	Matched z.b. ϵ (leerer String), `a`, `bab`, `bb`
`ba{3}b`	Matched z.B. `baaab`, `abaaaber`, `wabaaaben`, aber nicht `baaaab`
`ba{3,}b`	Matched z.B. `baaab`, `abaaaaber`, `abaaaaaber`, aber nicht `abaab`
`ba{2,3}b`	Matched z.B. `xyzbaablmn`, `xyzbaaablmn`
`(ab)*`	Matched `dddd`, `ddddab`, `abdddd`, `ddddababababbb`
`x(a|b)x`	Matched z.B. `dddxaxbbb`, `dddxbxbbb`, aber nicht `dddxcxbbb`.

Um in Python reguläre Ausdrücke benutzen zu können, muss das Modul `re` durch die Angabe `import re` importiert werden.

3.1 Übersicht über Metazeichen

3.1.1 Metazeichen . (Wildcard)

Dieses Metazeichen steht stellvertretend für genau ein Zeichen, aus der Menge aller vorkommenden atomaren Zeichen. Das newline Zeichen wird von diesem Metazeichen nicht gematched, außer es wird die FLAG `DOTALL` gesetzt.

3.1.2 Metazeichen [ ] (Buchstabenbereiche)

Buchstabenbereiche werden von den Metazeichen "Eckige Klammern" umschlossen und spezifizieren alle Buchstaben, die an dieser Stelle zugelassen sind. Es ist erlaubt, einen Bereich von Buchstaben anzugeben: `[a-f]` ist äquivalent zu `[abcdef]`. Dieser Bereich wird anhand der Codepoints der Zeichen bestimmt.

3.1.3 Metazeichen [^] (Negierte Buchstabenbereiche)

Negierte Buchstabenbereiche werden von den Metazeichen "Eckige Klammern" und dem ^-Zeichen als erstes Zeichen nach der öffnenden Klammer definiert. Sie spezifizieren alle Buchstaben, die an dieser Stelle nicht zugelassen sind.

3.1.4 Metazeichen * + ? { } (Quantifizierung)

Diese Metazeichen legen fest, wie oft das vorherige Zeichen (`a+`) oder die vorherige Zeichengruppierung (`(ab)+`) wiederholt wird. `[^aeiou]` findet z.B. alles was kein kleingeschriebener Vokal ist.

3.1.5 Metazeichen ^ (Anfang des Strings)

Soll ein (Sub)String am Anfang eines Strings gesucht werden, dann muss am Anfang des regulären Ausdrucks ein ^ stehen. Soll am Anfang jeder Zeile eines Strings gesucht werden, so muss die `MULTILINE` Flag gesetzt werden.

3.1.6 Metazeichen $ (Ende des Strings)

Soll ein (Sub)String am Ende eines Strings gesucht werden, dann muss am Ende des regulären Ausdrucks ein $ stehen. Soll am Ende jeder Zeile eines Strings gesucht werden, so muss das `MULTILINE` Flag gesetzt werden.

3.1.7 Metazeichen \ (Ausschalten der Metazeichenerkennung)

Soll in einem String eine Zeichenkette gefunden werden, die einen Metacharacter enthält, dann muss im regulären Ausdruck der gesuchte Metacharacter mit einem Backslash vor der Interpretierung als Metacharacter geschützt werden.

3.2 Das `re` Modul

Das `re` Modul definiert verschiedene Funktionen und Konstanten.

3.2.1 re.compile

Um einen regulären Ausdruck verwenden zu können, muss dieser zuerst kompiliert werden.

```
p = re.compile('tea')
```

`p` beschreibt nun das Muster (Pattern), in dem `t`, `e` und `a` direkt aufeinander folgen.

Das Verhalten eines regulären Ausdrucks kann mit Hilfe von Modifikatoren gesteuert werden.

Tabelle 3.3: Modifikatoren bei regulären Ausdrücken

Modifikator		Beschreibung
`re.I`	`re.IGNORECASE`	Achtet nicht auf Groß- und Kleinschreibung.
`re.L`	`re.LOCALE`	Das Verhalten von `\w`, `\W`, `\b` und `\B` wird an die aktuelle Locale angepasst.
`re.M`	`re.MULTILINE`	`^` und `$` beziehen sich auf Anfang/Ende einer Zeile, anstatt eines Strings.
`re.S`	`re.DOTALL`	`.` matched auch newline
`re.U`	`re.UNICODE`	Interpretiert Buchstaben nach dem Unicode Character Set. Beeinflusst Verhalten von `\w`, `\W`, `\b`, `\B`.
`re.X`	`re.VERBOSE`	Erlaubt "schönere" Reguläre Ausdrücke. Whitespace wird ignoriert (außer in einem Buchstabenbereich `[]`), `#` kann für Kommentare benutzt werden.

Mehrere Modifikatoren können mit `|` aneinander gereiht werden (`re.compile('spam', re.I|re.M)`).

```
p = re.compile('tea', re.I)
```

Jetzt beschreibt `p` das Muster, in dem `t`, `e` und `a` direkt aufeinander folgen und Groß-

und Kleinschreibung egal ist.

3.2.2 re.match

Die `match` Funktion vergleicht nur am Anfang eines Strings, ob der reguläre Ausdruck dort matched. Auch im `MULTILINE` Modus wird nur am Anfang des Strings verglichen, nicht am Anfang jeder Zeile.

Wir arbeiten mit folgendem Beispieltext:

```
text = "I don't like coffee. I drink TEA instead."
```

Nun verwenden wir `re.match`, um in `text` nach `p = re.compile('tea', re.I)` zu suchen.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> re.match(p, text)
>>>
```

Es passiert nichts. Um deutlicher zu machen was passiert, geben wir den Rückgabewert von `re.match(p, text)` mit `print` aus.

```
>>> print(re.match(p, text))
None
```

`re.match(p, text)` gibt `None` zurück, da das Muster `p` die Zeichen am Anfang des Strings nicht beschreibt.

Suchen wir allerdings nach folgendem Text:

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile("I don't", re.I)
>>> re.match(p, text)
>>>
```

Dann findet der reguläre Ausdruck den gesuchten Text am Textanfang und gibt aus, von wo bis wo der gesuchte String gefunden wird:

```
>>> print(re.match(p, text))
<_sre.SRE_Match object; span=(0, 7), match="I don't">
>>>
```

3.2.3 re.search

Die `search` Funktion verhält sich so wie die `match` Funktion, ist aber nicht auf den Beginn eines Strings beschränkt.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> re.search(p, text)
<_sre.SRE_Match object; span=(29, 32), match='TEA'>
```

`re.search` gibt ein Match Objekt zurück. Aus diesem kann entnommen werden, wo der Treffer gefunden wurde – span, Zeichen 29 (inklusive) bis 32 (exklusive), und auf welche Zeichen das Muster `p` zugetroffen hat – match.

Allerdings wurde hier nicht weitergeschaut, nachdem `re.search` den ersten Treffer gefunden hat. Das `tea` in `instead` wurde nicht gefunden.

3.2.4 re.finditer

Um alle Match Objekte für die Vorkommen von `p` in `text` zu bekommen, kann die `finditer` Funktion verwendet werden. Diese gibt einen Iterator zurück, der von links nach rechts Match Objekte für alle nicht überlappenden Vorkommen von `p` zurück gibt.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> for x in re.finditer(p, text):
...     x
...
<_sre.SRE_Match object; span=(29, 32), match='TEA'>
<_sre.SRE_Match object; span=(36, 39), match='tea'>
```

3.2.5 re.findall

Wenn man einfach nur alle gematchten Zeichenketten bekommen möchte, kann man dafür die `findall` Funktion verwenden. Zurückgebeben wird eine Liste von Strings.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> re.findall(p, text)
['TEA', 'tea']
```

3.2.6 re.sub

Mit der `sub` Funktion, können alle nicht überlappenden Vorkommen von `p` in `text` mit einem beliebigen String ersetzt werden.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> re.sub(p, 'xxx', text)
```

```
"I don't like coffee. I drink xxx insxxxd."
```

3.2.7 re.split

Mit der `split` Funktion, kann `text` an den Vorkommen von `p` in `text` gespalten werden. Zurückgegeben wird eine Liste von Strings.

```
>>> text = "I don't like coffee. I drink TEA instead."
>>> p = re.compile('tea', re.I)
>>> re.split(p, text)
["I don't like coffee. I drink ", ' ins', 'd.']
```

3.3 Zugriff auf gruppierte Teile des Regulären Ausdrucks

Man kann reguläre Ausdrücke mit runden Klammern strukturieren. Dadurch erreicht man erstens eine Gruppierung von Ausdrücken und zweitens einen Zugriff auf Teile des gesamten Treffers (Substrings).

```
>>> text = "What if cats could fly?"
>>> p = re.compile('What if (\w+) (.+)\?')
>>> m = re.search(p, text)
>>> m.group(1)
'cats'
>>> m.group(2)
'could fly'
```

3.4 Globales Suchen in einer Zeile

Standardmäßig (mit `re.match` bzw. `re.search`) sucht Python, genauso wie der UNIX-Befehl `grep`, nur nach dem ersten Vorkommen einer Zeichenkette innerhalb eines Strings. Kommt eine Zeichenkette mehrmals in einem String vor, dann übersieht Python die nachfolgenden Zeichenketten, da die Suchposition immer am Anfang der Zeile bleibt. Dieses Verhalten kann geändert werden indem man mit re.finditer einen Iterator benutzt, um über die Treffer zu iterieren. Findet Python in einer Zeile die gesuchte Zeichenkette, dann setzt Python in der Zeile hinter der gefundenen Zeichenkette die Suche fort. Treffer können sich also nicht überschneiden.

```
>>> text="what if cats could fly? the sky would be full of cats."
>>> p = re.compile('what if (\w+) (\w+) (\w+)\?')
>>> for m in re.finditer(p, text):
...     print(m.group(2), m.group(1), m.group(3))
...
```

```
could cats fly
```

3.5 Ersetzen von Strings mit re.sub

```
>>> text = "what if dogs would howl? then they are probably wolfs."
>>> p = re.compile('dogs')
>>> re.sub(p, 'xxx', text)
'what if xxx would howl? then they are probably wolfs.'
```

3.6 Greedy vs. nongreedy Verhalten bei Quantifizierung

Bei den `*` und `+` Quantifizierungen von Zeichen stellt sich die Frage, wie viele Zeichen Python in einem Pattern matched. Per Default versucht Python so viele Zeichen zu matchen wie möglich. Dieses Verhalten nennt man *Greedy-Matching*.

Greedy:

```
>>> text = "Schifffahrt"
>>> p = re.compile('f+')
>>> re.sub(p, 'x', text)
'Schixahrt'
```

`p` matched so viele `f` wie möglich und ersetzt diesen Match mit `x`.

```
>>> text = "Schifffahrt"
>>> p = re.compile('f+?')
>>> re.sub(p, 'x', text)
'Schixxxahrt'
```

`p` matched so wenig `f` wie nötig und ersetzt diese Matches mit `x`. Hier ist genau eins notwendig.

```
>>> text = "Schifffahrt"
>>> p = re.compile('f*?')
>>> re.sub(p, 'x', text)
'xSxcxhxixfxfxfxaxhxrxtx'
```

`p` matched so wenig `f` wie nötig und ersetzt diese Matches mit `x`. In diesem Fall sind gar keine `f` notwendig, da `f*?` auch den leeren String matched.

3.7 Zugriff auf gruppierte Teile des regulären Ausdrucks beim Ersetzen von Zeichenketten

Wie vorher erwähnt, kann man reguläre Ausdrücke mit runden Klammern strukturieren. Dadurch kann man beim Ersetzen von Zeichenketten auf die gefundenen Gruppen zugreifen. Python nummeriert die Gruppen durch und erlaubt, auf diese über Variablen (`\1`, `\2`, …) zuzugreifen.

```
>>> text = """Obama, Barack
... Merkel, Angela
... Roussef, Dilma
... Trudeau, Justin
... """
>>> p = re.compile('(\w+), (\w+)')
>>> print(re.sub(p, r'\2 \1', text))
Barack Obama
Angela Merkel
Dilma Roussef
Justin Trudeau
```

4 Integrierte Funktionen

4.1 abs

Die `abs` Funktion gibt den Betrag einer Zahl (Ganz- oder Fließkommezahl) zurück.

```
>>> abs(-1)
1
>>> abs(-5)
5
>>> abs(5)
5
>>> abs(-2.5)
2.5
>>> abs(2.5)
2.5
```

4.2 bool

Die `bool` Funktion gibt den Wahrheitswert des übergebenen Argumentes zurück.

```
>>> bool(1)
True
>>> bool([])
False
```

Jedes Objekt wird als `True` interpretiert, mit den folgenden Ausnahmen:

```
None
False
0       # Die Null jedes numerischen Typen
[], (), ""  # Eine leere Sequenz
```

4.3 chr

Die `chr` Funktion gibt genau das Zeichen als String zurück, dessen Unicode Code Point als Argument übergeben wird.

```
>>> chr(65)
'A'
>>> chr(120)
'x'
```

4.4 dict

Die `dict` Funktion erzeugt ein neues Dictionary. Dabei kann entweder ohne Argument ein leeres Dictionary erzeugt werden, über eine Liste von 2-Tupeln ein neues Dictionary erstellt werden, oder mit Keyword-Argumenten gearbeitet werden:

```
>>> leeres_dict = dict()
>>> neues_dict = dict([("a",1),("b",2),("c",3)])
>>> print(neues_dict)
{'c': 3, 'a': 1, 'b': 2}
>>> neues_dict = dict(a=1, b=2, c=3)
>>> print(neues_dict)
{'c': 3, 'a': 1, 'b': 2}
```

4.5 float

Die `float` Funktion erzeugt eine Zahl vom Datentypen Float. Der Funktion können alle numerischen Datentypen ausser `complex` übergeben werden.

```
>>> float()
0.0
>>> float(55)
55.0
```

4.6 input

Die `input` Funktion liest eine Eingabe vom Benutzer ein und gibt sie als String zurück. Der Parameter ist dabei optional und wird dem Benutzer als Eingabeaufforderung angezeigt.

```
eingabe = input()
eingabe_zahl = int(input("Bitte geben Sie eine Ganzzahl ein: "))
```

4.7 int

Die `int` Funktion gibt einen Integer Wert für das übergebene Argument zurück. Mit dieser Funktion können z.B. Strings, die eine Zahl darstellen, oder Floats in Integer umgewandelt werden. Bei einer Fließkommazahl wird auf die nächste Ganzzahl abgerundet.

```
>>> int('255')
255
>>> int(1.99)
1
```

4.8 join

Die `join` Methode konkateniert String Elemente einer Liste. Der Seperator zwischen diesen Elementen ist der String welcher diese Methode zur Verfügung stellt.

```
>>> names = ["Klaus", "Jonas", "Amelie", "Rosa"]
>>> sep = ", "
>>> sep.join(names)
'Klaus, Jonas, Amelie, Rosa'
>>> sep = "\n"
>>> sep.join(names)
'Klaus\nJonas\nAmelie\nRosa'
>>> print(sep.join(names))
Klaus
Jonas
Amelie
Rosa
```

4.9 len

Die `len` Funktion gibt die Länge (Anzahl der Elemente) eines sequenziellen Objektes zurück.

```
>>> len("Hello World")
11
>>> len([4, 6, 2, 8, 4, 7])
6
```

4.10 list

Die `list` Funktion kann dazu verwendet werden, um ein beliebiges sequenzielles Objekt in eine Liste umzuwandeln.

```
>>> list()
[]
>>> list((1,2,3))
[1,2,3]
>>> list({"a": 1, "b": 10})
["a","b"]
>>> list(range(2,12,3))
[2,5,8,11]
```

4.11 max

Die `max` Funktion gibt das größte der übergebenen Argumente zurück. Oder, wenn nur ein (sequenzielles!) Argument übergeben wird, das größte enthaltene Element.

```
>>> max(7, -1, 9, 5)
9
>>> max('a', 'b', 'c')
'c'
>>> max([7, -1, 9, 5])
9
```

4.12 min

Die `min` Funktion gibt das kleinste der übergebenen Argumente zurück. Oder, wenn nur ein (sequenzielles!) Argument übergeben wird, das kleinste enthaltene Element.

```
>>> min(7, -1, 9, 5)
-1
>>> min('a', 'b', 'c')
'a'
>>> min([7, -1, 9, 5])
-1
```

4.13 open

Die `open` Funktion öffnet eine Datei und gibt ein Datei-Objekt zurück. Der Path der zu öffnenden Datei wird als String angegeben. Zusätzlich darf der Modus, in dem die Datei geöffnet werden soll entweder unbenannt als zweites Argument, oder als benanntes Argument angegeben werden. Das optionale benannte Argument `encoding` gibt das zu verwendende Encoding an, wobei ansonsten das vom System als Standard betrachtete Encoding verwendet wird.

```
>>> file = open('file.txt')
>>> datei = open('otherfile.txt','w')
>>> quelle = open('quelldatei.txt', mode='r')
>>> text = open('samefile.txt', 'a', encoding='UTF-8')
>>> ziel = open('zieldatei.txt', mode='w', encoding='Latin-1' )
```

4.14 ord

Die `ord` Funktion bekommt ein String mit einem Unicode-Zeichen als Argument gegeben, und gibt den Integer, der dieses Zeichen repräsentiert, zurück.

```
>>> ord('A')
65
>>> ord('Z')
90
>>> ord('`')
96
>>> ord('a')
97
```

4.15 pow

Die `pow` Funktion bekommt zwei oder drei Argumente gegeben, und gibt das Ergebnis des ersten Arguments potenziert mit dem zweiten zurück. Wenn das optionale dritte Argument vorhanden ist, so wird noch der Modulo mit diesem Argument berechnet. Die Variante mit den zwei Argumenten ist äquivalent zum "power operator": `x**y`.

4.16 print

Die `print` Funktion "druckt" Objekte standardmäßig auf die Standardausgabe. Wohin gedruckt werden soll kann mit dem Keyword Argument `file` geändert werden. Mit dem

Keyword Argument `sep` (default: `' '`) kann der Seperator bestimmt werden, der beim Drucken von mehreren Strings diese trennt. Mit dem Keyword Argument `end` (default: `\n`) kann bestimmt werden, was nach dem Drucken der Objekte noch zusätzlich gedruckt werden soll.

```
>>> print("hello", "world")
hello world
>>> print("hello", "world", sep='-')
hello-world
>>> print("hello", "world", end='XXX')
hello worldXXX>>>
>>> print("hello", "world", sep='_', end='.py\n')
hello_world.py
```

4.17 range

Die `range` Funktion erzeugt eine nachträglich unveränderliche "Liste" von Zahlenwerten. Die `range` Funtion wird im Detail im Kapitel über die for-Schleife erklärt.

4.18 reversed

Die `reversed` Funktion bekommt ein sequenzielles Objekt als Argument gegeben und gibt einen invertierten Iterator zurück.

```
>>> reversed([1,2,3,4,5])
<list_reverseiterator object at 0x107262eb8>
>>> for n in reversed([1,2,3,4,5]):
...     print(n)
...
5
4
3
2
1
```

4.19 round

Die `round` Funktion gibt den Integer zurück, der dem gegebenen (ersten) Argument am nächsten ist. Ein optionales zweites Argument kann benutzt werden, um nicht zum nächsten Integer zu runden, sondern zu einer bestimmten Stelle nach dem Dezimalpunkt.

```
>>> round(23.3)
23
>>> round(23.123456789, 1)
23.1
>>> round(23.123456789, 4)
23.1235
>>> round(23.123456789, 8)
23.12345679
```

4.20 sorted

Die `sorted` Funktion nimmt ein iterierbares Objekt (wie zum Beispiel eine Liste) und gibt eine neue sortierte Liste zurück. Mit dem Keyword Argument `key` kann eine Funktion angegeben werden, die vor der Vergleichsoperation auf die Elemente angewendet werden soll. Mit dem boolschen Keyword Argument `reverse` kann angegeben werden, ob die sortierte Liste invertiert zurückgegeben werden soll.

```
# Natürliche Ordnung (Zahlen)
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
# Natürliche Ordnung (Zeichencodes)
>>> sorted("This is a test string from Andrew".split())
['Andrew', 'This', 'a', 'from', 'is', 'string', 'test']
# Benannte Key-Funktion
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
# Anonyme Key-Funktion
>>> sorted("This is a test string from Andrew".split(), key = lambda word: word.lower())
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
# Key-Funktion und Reverse
>>> sorted("This is a test string from Andrew".split(), key=str.lower, reverse=True)
['This', 'test', 'string', 'is', 'from', 'Andrew', 'a']
```

Es ist auch möglich die `sort()` Funktion zu verwenden, die von Listen bereitgestellt wird.

```
>>> numbers = [5, 2, 3, 1, 4]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4, 5]
```

Hierbei ist zu beachten, dass sort() keinen Rückgabewert hat, sondern die zugrundeliegende Datestruktur modifiziert.

4.21 sorted bei Dictionaries

Da Dictionaries ebenfalls iterierbare Elemente sind, können sie auch mit der `sorted` Funktion sortiert werden. Allerdings muss man bei Dictionaries beachten, ob man nach den `keys` oder dem `values` sortieren möchte. Mit dem Funktionsargument `key` kann eine Funktion angegeben werden, die vor der Vergleichsoperation auf die Elemente angewendet werden soll. Zusätzlich kann mit dem boolschen Keyword Argument `reverse` angegeben werden, ob die Paare nach der Frequenz absteigend sortiert werden sollen.

```
# gegeben eine Frequenzliste mit Paaren <word> <frequenz>
# Sortiert absteigend die Paare nach der Freqenz
>>> freq_sorted = sorted(frequenzliste.items(), key=lambda x: x[1],reverse=True):
```

4.22 str

Die `str` Funktion gibt eine String-Version des übergebenen Arguments zurück.

```
>>> 1
1
>>> str(1)
'1'
```

4.23 sum

Die `sum` Funktion gibt die Summe der Elemente eines iterierbaren Objektes zurück. Mit einem optionalen zweiten Argument kann eine Zahl festgelegt werden, die zusätzlich addiert werden soll.

```
>>> sum([1,2,3])
6
>>> sum([1,2,3], 10)
16
```

4.24 tuple

Die `tuple` Funktion erzeugt ein Tuple aus einem Sequenziellen Objekt.

```
>>> tuple()
()
>>> tuple([1,2,3,4,5])
(1,2,3,4,5)
```

5 Funktionen

5.1 Modulare Programmierung und Unterprogramme

Um große Programmieraufgaben übersichtlich und strukturiert zu realisieren, ist es sehr empfehlenswert, Teilaufgaben des Programms in eigenen Programmteilen, sogenannten Unterprogrammen, realisieren zu lassen. Unterprogramme können selbstständig entwickelt, getestet, optimiert, von Fehlern befreit und in anderen Programmen wiederverwendet werden. Die Unterprogrammtechnik ist auch das zentrale Konzept bei der modularen Programmierung: Größere Unteraufgaben werden in eigenen Modulen erledigt.

Eine Programmiersprache, die die Unterprogrammtechnik unterstützt, muss in der Lage sein, Programmteile zusammenzufassen, diesen Programmteilen einen Namen zu geben und ihnen Werte zu übergeben, bzw. von ihnen Werte ermitteln zu lassen, die daraufhin im aufrufenden Programm zur Verfügung stehen.

Python unterstützt die Unterprogrammtechnik, Programmteile können in Unterprogrammen, genannt Funktionen, zusammengefasst und wiederverwendet werden. An Funktionen können Werte übergeben werden, die Werte können innerhalb der Funktion weiterverarbeitet werden und das Ergebnis kann dem aufrufenden Programm zurückgegeben werden. Das Unterprogramm kann somit eine eigene Aufgabe realisieren und mit dem aufrufenden Programm zusammenarbeiten.

Ein Beispiel für Funktionen in Python ist z.B. die Funktion `print` zum Ausgeben von Text.

```
print("Hallo Welt!")
```

Der Funktion print wird das Argument `"Hallo Welt!"` übergeben: Im Unterprogramm print wiederum werden alle notwendigen Schritte ausgeführt, die auf dem Bildschirm genau an der richtigen Stelle den Text “Hallo Welt!” ausgeben.

5.2 Funktionen

Bereits aus der Mathematik bekannt ist der Begriff der Funktion. Die Funktion `f(x) = x²` ordnet zum Beispiel jedem Parameter `x` seinen quadrierten Funktionswert zu. Eine Funktion im mathematischen Sinne besteht also aus einem Namen, einer Liste von Parametern und einer Rechenvorschrift für den Funktionswert.

Dieses Konzept findet sich auch in Python wieder. Betrachten Sie beispielsweise die eingebaute Funktion `len`, die als Argument eine iterierbare Sequenz erhält:

Beispiel: Funktion `len` im interaktiven Modus

```
>>> len("Dieser String ist das Argument")
30
```

Offensichtlich besteht auch eine Funktion in Python aus einem Funktionsnamen (`len`), einer Menge von Parametern, die mit Argumenten versorgt werden (im Beispiel: ein String) und einem Rumpf, der die auszuführenden Rechenanweisungen der Funktion (genannt *Funktionskörper*) enthält und das Ergebnis der Funktion berechnet.

Eine Funktion in Python kann, so wie `len`, einen *Rückgabewert* haben, muss dies aber nicht explizit: Die Funktion `print` zum Beispiel hat keinen, also `None`.

Ein großer Vorteil bei der Verwendung von Funktionen ist die Vermeidung von Redundanz: Man stelle sich z.B. vor, man müsste den Programmcode hinter `print` jedes mal von Hand ausschreiben, wenn man Text auf die Konsole ausgeben möchte…

5.3 Definition von Funktionen in Python

Python bietet eine ganze Reihe eingebauter Funktionen, und über das Einbinden von Modulen können unzählige weitere hinzugewonnen werden. Dennoch werden diese Funktionen zum Schreiben einer eigenen Programmaufgabe nicht ausreichen: Eigene Funktionen müssen definiert werden.

Eine Funktionsdefinition muss über drei Dinge verfügen:

- Eine Funktion muss einen *Namen* haben, über den sie in anderen Teilen des Programmes eindeutig aufgerufen werden kann. Zur Formatierung von Funktionsnamen gelten die gleichen Leitlinien wie für die Formatierung von Variablennamen.
- Eine Funktion muss eine *Schnittstelle* haben, über die Informationen vom aufrufenden Programmteil an die Funktion übertragen werden. Eine Schnittstelle kann aus beliebig vielen (unter Umständen auch keinen) *Parametern* bestehen. Innerhalb der Funktion wird jedem dieser Parameter ein Name gegeben, unter dem sie wie Referenzen im Funktionskörper verwendet werden können.
- Eine Funktion muss einen *Wert* zurückgeben. Jede Funktion gibt automatisch `None` zurück, wenn der Rückgabewert nicht explizit angegeben wurde.
- Das Schüsselwort zum Definieren einer Funktion ist `def`

Beispiel: Definition einer Funktion

```
def mein_Funktionsname(parameter_1, parameter_2, parameter_n):
    Anweisung1
    Anweisung2
    AnweisungN
```

5.4 Datentausch zwischen Funktion und Hauptprogramm

Damit eine Funktion ihre Arbeit verrichten kann, müssen ihr beim Funktionsaufruf einige Informationen mitgegeben werden, die sogenannten *Argumente*.

> Namensgebung: *Parameter* sind die Platzhalter für die tatsächlichen *Argumente*

Bei der Definition einer Funktion wird bestimmt, welche Argumente der Funktion übergeben werden sollen. Als Beispiel soll hier eine Funktion mit Namen `summe` dienen, die zwei Argumente `a` und `b` verlangt:

```
def summe(a,b):
    return a + b
```

Diese einfache Funktion berechnet die Summe der zwei Argumente, die ihr übergeben werden:

```
>>> summe(2,4)          # Funktionsaufruf
6
>>> meine_zahl = summe(7,13)
>>> print(meine_zahl)
20
```

> Formatierungskonvention: Lassen Sie zwischen Funktionsnamen und Argumenten kein Leerzeichen!

Dass dem aufrufenden Programm das Ergebnis der Berechnung überhaupt zur Verfügung steht, ist dem Schlüsselwort `return` zu verdanken: Es weist die Funktion an, dass an dieser Stelle die Funktion sofort verlassen wird, um den nebenstehenden Ausdruck hinter `return` an das Hauptprogramm zurückzugeben. Betrachtet man einen Funktionsaufruf vom aufrufenden Programm aus, dann kann man sich das so vorstellen, als ob der Funktionsaufruf “durch den Rückgabewert ersetzt wird”:

```
>>> ergebnis = summe(5,5)
# "wird zu" ergebnis = 10
>>> print(ergebnis)
10
```

Wird das `return`-Statement ohne Argument aufgerufen oder nicht verwendet, so ist der Rückgabewert der Funktion `None` - ein spezieller Datentyp, der anzeigt, dass "nichts" zurückgegeben wurde.

Folglich sind folgende zwei Funktionen effektiv identisch:

```
def funktion1():
    # Do nothing
    return

def funktion2():
    # Do nothing

# Test:
>>> a = funktion1()
>>> b = funktion2()
>>> a
None
>>> b
None
```

5.4.1 Optionale Funktionsparameter

Angenommen, man möchte an die Funktion `summe` nicht nur zwei Argumente übergeben, sondern hat drei oder vier Zahlen, die man addieren möchte. Nach der bisherigen Syntax müsste man zwei neue Funktionen schreiben, die jeweils drei oder vier Argumente akzeptieren. Um dieses Problem zu umgehen bietet Python die Mögichkeit *Optionale Parameter* anzugeben, bei denen es dem Aufrufenden freigestellt ist, sie mit Argumenten zu belegen oder nicht. Damit es zu keinen Problemen kommt, wenn die Funktion mit weniger als ihrer maximalen Anzahl an Argumenten aufgerufen wird, wird diesen Parametern ein *Default-Wert* zugeordnet, den diese Parameter annehmen, sollten sie nicht explizit mit einem Wert belegt werden.

Parameter werden automatisch *optional*, wenn ihnen bei der Definition der Funktion ein Default-Wert zugeordnet wird.

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

Die Funktion kann jetzt mit 2, 3 oder 4 Argumenten aufgerufen werden:

```
>>> summe(4,16)
20
>>> summe(-2,7,4)
9
```

```
>>> summe(10,6,22,1)
39
```

5.4.2 Beliebige Anzahl von Parametern

Oft macht es sogar Sinn, wenn eine Funktion eine beliebige Anzahl an Parametern annimmt: Die Summe-Funktion von oben könnte genausogut auch mit 5, 6, oder 100 Zahlen arbeiten.

Python stellt zu diesem Zweck eine besondere Notation zur Verfügung, mit der sich der "endlose" Parameter markieren lässt:

```
def summe(*summanden):
    ergebnis = 0
    for summand in summanden:
        ergebnis = ergebnis + summand
    return ergebnis
```

Die übergebenen Parameter werden als ein n-Tupel an die Funktion übergeben, wie folgendes Beispiel zeigt:

Beispiel: Parameterübergabe als Tupel

```
def testfunktion(*parameter):
    return parameter
# Test:
>>> print(testfunktion(1,2))
(1,2)
```

5.5 Probleme bei der Übergabe von Argumenten an eine Funktion

In Python werden bei einem Funktionsaufruf nicht die Werte der Argumente an die Parameter übergeben, sondern es wird funktionsintern mit Referenzen, also "Zeigern", auf die Argumente gearbeitet. Diese Methode, Argumente zu übergeben, nennt man auch *Call by Reference*.

Der Vorteil dieses Verfahrens liegt im Speicherbedarf und der Geschwindigkeit des Programmes begründet: Wenn zum Beispiel einer Funktion ein großes Dictionary übergeben wird, müsse der gesamte Inhalt des Dictionaries in die Funktion kopiert werden. Dies würde sowohl signifikant mehr Speicherplatz als auch zusätzliche Rechenzeit zum Erstellen der Kopie bedeuten - auch, wenn eine Kopie auf Grund der Programmlogik gar nicht nötig wäre.

Die grundsätzliche Übergabe der Argumenten als Referenz an eine Funktion verdient allerdings besondere Beachtung bei Python, da Python bei der Arbeit mit Daten zwei Arten von Datentypen unterscheidet:
Mutable-Datentypen (Liste oder Dictionary) und *Immutable*-Datentypen (Zahlen, Strings). Bei *Mutable*-Datentypen werden Änderungen seiner Werte immer auf dem identischen Speicherbereich durchgeführt, bei *Immutable*-Datentypen wird immer ein neues Objekt im Speicher angelegt, das den neuen Wert trägt.

Bei Argumenten unterscheidet Python ebenso: Sind Datentypen der Argumente *Mutable* oder *Immutable*?
Werden *Mutable* Argumente übergeben, dann bewirkt jede Änderung der Werte dieser Argumente, dass die Funktion auf dem gleichen Speicherbereich wie das aufrufende Programm arbeitet: Das als Argument übergebene Objekt **wird verändert**.

Werden *Immutable* Argumente übergeben, dann wird jede Änderung der Werte immer auf einem neuen Objekt aufgeführt. Das übergebene Objekt wird also **nicht** geändert.

Beachtet der Programmierer diese Unterscheidung nicht, dann kann es schnell zu sogenannten (ungewünschten) *Side Effects* kommen, wenn auf diese Weise ein *Mutable*-Datentyp (Liste oder Dictionary) übergeben wird:

Beispiel: Seiteneffekt bei *Mutable* Argumenten

```
def f(liste):
    liste += [6,7,8,9]

# Test
>>> zahlen = [1,2,3,4,5] # Listen sind 'mutable'
>>> print(zahlen)
[1,2,3,4,5]
>>> f(zahlen)
>>> print(zahlen)
[1,2,3,4,5,6,7,8,9]
```

Es wird in diesem Beispiel zuerst eine Funktion definiert, die eine Liste (von Zahlen) erwartet und sie im Körper der Funktion verändert, aber keinen Rückgabewert hat. Im Hauptprogramm wird die Liste angelegt und ausgegeben. Danach wird die Funktion ausgeführt und die Liste erneut ausgegeben.

Es fällt auf: Obwohl die Funktion keinen Rückgabewert hat, der in eine Variable gespeichert worden wäre, hat sich der Wert der Variable ‘zahlen’ verändert: Die Änderung hat sich nicht auf den Kontext der Funktion beschränkt, es ist ein *Side Effect* aufgetreten, weil die Liste ein *Mutable*-Datentyp ist. *Mutable* heißt, dass das Objekt veränderlich ist.

Sofern dieses Verhalten nicht ausdrücklich gewollt ist, sollten Sie bei der Parameterübergabe eine Kopie der Liste oder des Dictionarys erzeugen. Python stellt uns hierfür über das Slicing einen einfachen Operator zur Verfügung, der in diesem Fall festgelegt, das für dieses Argument eine Kopie übergeben wird: `[:]`

Im Falle der Liste könnte das zum Beispiel so aussehen:

Beispiel: Explizite Übergabe einer Kopie eines *mutable* Datentyps Liste:

```
def f(liste):
    liste += [6,7,8,9]

# Test
>>> zahlen = [1,2,3,4,5]    # Listen sind 'mutable'
>>> print(zahlen)
[1,2,3,4,5]
>>> f(zahlen[:])        # Erzeugen einer Kopie mittels 'Slicing'
>>> print(zahlen)
[1,2,3,4,5]
```

Natürlich ist es ebenfalls möglich, die Kopie innerhalb der Funktion zu erstellen.

Eine seltenere Form von Seiteneffekten kann auftreten, wenn (optionalen) Parametern einer Funktion ein *Default-Wert* zugeordnet ist, der *Mutable* ist:

Beispiel: Seiteneffekt bei Default-Werten

```
def f(liste=[a,b,c]):
    liste += [d,e]
    print(liste)

# Test
>>> f()
[a,b,c,d,e]
>>> f()
[a,b,c,d,e,d,e]
>>> f()
[a,b,c,d,e,d,e,d,e]
# ...
```

Wie an diesem Beispiel ersichtlich ist, bleibt der Default-Wert des Funktionsparameters `liste` zwischen Funktionsaufrufen bestehen, er wird also *nicht* bei jedem Funktionsaufruf neu angelegt. Das hat zur Folge, dass Änderungen an dem Objekt des Default-Wertes über mehrere Funktionsaufrufe hinweg bestehen bleiben.

Sie sollten deshalb darauf verzichten, Instanzen von *Mutable*-Datentypen (Liste und Dictionary) als Default-Wert zu verwenden. Diesen *Side-Effect* kann man zum Beispiel umgehen, indem man stattdessen im Funktionskörper bei jedem Aufruf, der den Default-Wert nutzt, eine neue Instanz erzeugt:

Beispiel: Default-Wert `None`

```
def f(liste=None):
    if liste is None:
```

```
liste = [a,b,c]
```

`None` ist *immutable*, kann also bedenkenlos verwendet werden.

5.6 Gültigkeitsbereich von Variablen

Wie bereits bei der Behandlung von Variablen festgestellt wurde, ist der Gültigkeitsbereich von Variablen begrenzt, abhängig davon, an welcher Stelle sie eingeführt werden. Der Gültigkeitsbereich der Variablen wird eingeschränkt auf den Block, innerhalb dessen sie deklariert worden ist. Nur innerhalb dieses Blockes kann schreibend oder lesend auf sie zugegriffen werden. Lokale Variablen in einem Funktionsblock sind nur innerhalb dieses Funktionsblocks verwendbar. Es kann z.B. auf eine Variable, die in einem Funktionsblock eingeführt wird, im Hauptprogramm nicht zugegriffen werden.

Hierbei wird im Grunde zwischen zwei Arten von sogenannten *Namespaces* oder *Namensräumen* unterschieden:

- Der *globale* Namensraum
- Der/Ein *lokaler* Namensraum

Der globale Namensraum ist die oberste Ebene: Alle Funktionen und Variablen, die im Hauptprogramm definiert werden, sind Teil dieses Namensraumes und sind auch in allen lokalen Namensräumen gültig.

Jede Funktion dagegen besitzt ihren eigenen lokalen Namensraum, dessen Gültigkeit auf den Funktionskörper beschränkt ist.

Aus diesem Grund ist es möglich gleichnamige Variablen sowohl innerhalb von Funktionsdefinitionen als auch im Hauptprogramm zu nutzen: Muss der Python-Interpreter entscheiden, von welcher Referenz er anhand eines Variablennamens *lesen* soll, so wird zuerst überprüft, ob es eine lokale Variable mit diesem Namen gibt, und erst danach auf die globale Variable zurückgegriffen. Dies soll an folgendem Beispiel deutlich werden:

Beispiel: Aus lokaler Variable lesen

```
var = "globale Variable"

def funktion(var):
    print(var)

# Test
>>> print(var)
'globale Variable'
>>> funktion("lokale Variable")
'lokale Variable'
```

```
>>> print(var)
'globale Variable'
```

Die beiden Variablen mit Namen `var` haben nichts miteinander zu tun: Sie existieren in unterschiedlichen Namensräumen.

Aufpassen muss der Programmierer, wenn er innerhalb eines lokalen Namensraumes auf eine globale Variable *schreibend* zugreifen will. Wenn nicht anders angegeben, wird in diesem Fall nämlich automatisch stattdessen eine gleichnamige lokale Variable angelegt:

Beispiel: In lokale Variable schreiben

```
var = "globale Variable"

def funktion(arg):
    var = arg[:]
    print(var)

#Test
>>> print(var)
'globale Variable'
>>> funktion("neuer Text für var")
'neuer Text für var'
>>> print(var)
'globale Variable' # Die globale Variable var wurde nicht überschrieben
```

In diesem Beispiel existiert zu Beginn des Funktionsaufrufes eine globale Variable mit Namen `var`. Doch in der Zuweisung der Kopie von `arg` an die Variable `var` wird eine lokale Variable dieses Namens angelegt.

Will man jedoch an dieser Stelle explizit auf die globale Variable zugreifen, so bietet Python ein Schlüsselwort, um dies dem Interpreter mitzuteilen:

```
global
```

Achtung: mit globalen statt lokalen Variablen zu arbeiten wird oft als schlechter Stil angesehen, weil man damit auf globaler Ebene die Hoheit über seine Variablen abgibt. Gerade bei Projekten, an denen mehrere Programmierer arbeiten, kann das zu großen Problemen führen.

Beispiel: Das Schlüsselwort `global`

```
var = "globale Variable"

def funktion(arg):
    global var  # "Benutze die globale Variable und lege keine lokale an"
    var = arg[:]
    print(var)
```

```
# Test
>>> print(var)
'globale Variable'
>>> funktion("neuer Text für var")
'neuer Text für var'
>>> print(var)
'neuer Text für var' # Die globale Variable var *wurde* überschrieben
```

Zusammenfassung

- Jede Variable (und Funktion) existiert in einem bestimmten Namensraum
- Innerhalb eines lokalen Namensraumes kann jederzeit *lesend* auf globale Variablen zugegriffen werden
- Wird versucht, in einem lokalen Namensraum *schreibend* auf eine globale Variable zuzugreifen, so wird stattdessen eine neue lokale Variable angelegt
- Mittels des Schlüsselwortes `gobal` kann explizit auf eine globale Variable verwiesen werden

5.7 Beispiele von Funktionsaufrufen mit mutable und immutable Argumenten

Beispiel: Arbeit mit einer Variblen im Namespace der Funktion (lokale Variable)

```
def inc(i):
## die Variable i ist eine lokale Variable und
## gehört zum Namespace der Function inc
     i = i + 1
     return i

# Wir definieren ein ganzzahliges Objekt mit dem Wert 1
# Die Variable i zeigt drauf:
i=1

# Wir definieren ein ganzzahliges Objekt mit dem Wert -4
# Die Variable ii zeigt drauf:
ii=-4
print("i=",i,"Adress before function call of i=",hex(id(i)))
print("ii=",ii,"Adress before function call of ii=",hex(id(ii)))

ii=inc(i)

## am Objekt auf das i zeigt ändert sich nichts, i behält Adresse
```

```
print("i=",i,"Adress after function call of i=",hex(id(i)))
## ii bekommt neuen Wert und erzeugt neues Objekt, ii muss darauf zeigen
## Variable ii braucht neue Adresse
print("ii=",ii,"Adress after function call of ii=",hex(id(ii)))

# Wir definieren ein ganzzahliges Objekt mit dem Wert 3
# Die Variablen x und y zeigen drauf:
x=y=3
print("Adress of x=",hex(id(x)))
print("Adress of y=",hex(id(y)))

# Wir definieren ein ganzzahliges Objekt mit dem Wert 4, y muss drauf zeigen
# Variable y braucht neue Adresse
y=4;
print("x=",x,"y=",y)
print("Adress of x=",hex(id(x)))
print("Adress of y=",hex(id(y)))
```

Beispiel: Arbeit auf einem Mutable List Argument

```
def incall(all):
   for i in range(0,len(all)-1):
     all[i] += 1


# Wir definieren ein mutable List_object den Werte [1,2,3,4]
# Die Variable values zeigt drauf:
values = [1,2,3,4]

print("values=",values,"Adress before function call of values=",hex(id(values)))
incall(values)
## das Objekt ist mutable, die Werte ändern sich,
## values behält aber Adresse
print("values=",values,"Adress after function call of i=",hex(id(values)))
```

Beispiel: Arbeit mit einer Liste, auf die zwei Variablen zeigen

```
def incall(all):
   for i in range(0,len(all)-1):
     all[i] += 1

# Wir definieren ein mutable List_object den Werte [1,2,3,4]
# Die Variable values zeigt drauf:
values = [1,2,3,4]
# Die Variable values_2 zeigt ebenfalls drauf:
values_2 = values;
```

```
print("values=",values,"Adress before function call of values=",hex(id(values)))
incall(values)
## das Objekt ist mutable, die Werte ändern sich, values behält aber Adresse
print("values=",values,"Adress after function call of i=",hex(id(values)))

## das Objekt ist mutable, die Werte ändern sich,
## Die Verbindung von values und values_2 bleibt bestehen
## Values_2 behält die Adresse
print("values_2=",values_2,"Adress after function call of i=",hex(id(values_2)))
```

Beispiel: Ein Mutable List Element wird als Kopie übergeben (Version 1)

```
# Beispiel immutable Variable,
def incall(all):
   for i in range(0,len(all)-1):
     all[i] += 1

# Wir definieren ein mutable List_object den Werte [1,2,3,4]
# Die Variable values zeigt drauf:
values = [1,2,3,4]
# Die Variable values_2 zeigt ebenfalls drauf:
values_2 = values;

print("values=",values,"Adress before function call of values=",hex(id(values)))
## Übergabe der Argumente in einer Kopie
incall(values[:])
## Das Objekt ist zwar mutable, es wurden aber kopierte Werte übergeben
## Die Werte haben sich nicht geändert und auch nicht die Adresse
print("values=",values,"Adress after function call of i=",hex(id(values)))
```

Beispiel: Ein Mutable List Element wird als Kopie übergeben (Version 2)

```
# Beispiel immutable Variable,
#
def incall(all):
   tmpall = all[:]
   for i in range(0,len(tmpall)-1):
     tmpall[i] += 1

# Wir definieren ein mutable List_object den Werte [1,2,3,4]
# Die Variable values zeigt drauf:
values = [1,2,3,4]
# Die Variable values_2 zeigt ebenfalls drauf:
values_2 = values;
```

```
print("values=",values,"Adress before function call of values=",hex(id(values)))
## Übergabe der Argumente in einer Kopie
incall(values)
## Das Objekt ist zwar mutable, es wurden aber kopierte Werte übergeben
## Die Werte haben sich nicht geändert und auch nicht die Adresse
print("values=",values,"Adress after function call of i=",hex(id(values)))
```

5.8 Rekursion

Das Wort "Rekursion" stammt aus dem Lateinischen, das Verb "recurrere" bedeutet soviel wie "zurücklaufen" oder "wiederkehren". Die Mathematik spricht von einer rekursiven Funktion, wenn sie sich durch sich selbst definiert, also zum abarbeiten der Funktion zur Funktion selber "zurückgelaufen" wird:

$$f(n) = n * f(n-1), \text{ falls } n > 1, \text{ sonst } 1$$

In der Informatik ist das Konzept der Rekursion sehr nahe am mathematischen Ursprung: Eine Funktion wird rekursiv genannt, wenn sie sich selbst ein oder mehrmals in ihrem Funktionskörper aufruft. Außerdem muss die Rekursion *terminieren*, also einen Endpunkt besitzen, da sie sonst eine Endlosschleife darstellt. Deswegen braucht jede rekursive Funktion eine *Abbruchbedingung*. Im obrigen Beispiel der Fakultät ist die Abbruchbedingung `f(1) = 1`.

Das zugrundeliegende Prinzip ist etwas ur-menschliches: Das vorliegende Problem wird solange rekursiv in kleinere Teilprobleme zerlegt, bis es trivial wird. Aus den entstandenen Teillösungen wird anschließend das Gesamtergebnis zusammengebaut.

Beispiel: Teillösungen der Rekursion

```
f(5) = 5 * f(4)                                  #  | Problem in Teilprobleme zerlegen
           f(4) = 4 * f(3)                         #  |
                      f(3) = 3 * f(2)              #  |
                                 f(2) = 2 * f(1)   #  V
                                            f(1) = 1 #  <- Abbruchbedingung
                                       = 2 * 1      #  | Ergebnis zusammenbauen
                            = 3 * 2                 #  |
                  = 4 * 6                           #  |
    = 5 * 24                                        #  |
= 120                                          #  V
```

Damit dieses Prinzip in einer Programmiersprache umsetzbar ist, muss diese Sprache über einen Mechanismus verfügen, der diese Zwischenergebnisse und neuen Aufrufe der Funktion verwalten kann. Glücklicherweise verfügt Python über einen *Stack* ("Stapelspeicher"), um diese Informationen verarbeiten zu können. Die maximale *Rekursionstiefe*, also die maximale Anzahl an verschachtelten Funktionsaufrufen, ist dabei durch die

Laufzeitumgebung vorgegeben, sollte aber nur in Extremfällen zum Vorschein treten.

Die Implementierung in Python ist beinahe so kurz wie die Mathematische Definition:

Beispiel: Fakultätsfunktion in Python

```
def fakultät(n):
    if n == 1:
        return 1
    else:
        return n * fakultät(n-1)
```

Um zu sehen, was bei einem Aufruf dieser Funktion genau passiert, wird das obige Beispiel im Folgenden um einige Zeilen Textausgabe erweitert:

Beispiel: Fakultätsfunktion in Python mit Textausgabe

```
def fakultät(n):
    print("Fakultät(n) wurde aufgerufen mit n = " + str(n))
    if n == 1:
        return 1
    else:
        erg = n * fakultät(n-1)
        print("Zwischenergebnis für " + str(n) + " * fakultät(" + str(n-1) + "): " + str(er
        return erg

# Test
>>> fakultät(5)
Fakultät(n) wurde aufgerufen mit n = 5
Fakultät(n) wurde aufgerufen mit n = 4
Fakultät(n) wurde aufgerufen mit n = 3
Fakultät(n) wurde aufgerufen mit n = 2
Fakultät(n) wurde aufgerufen mit n = 1
Zwischenergebnis für 2 * fakultät(1): 2
Zwischenergebnis für 3 * fakultät(2): 6
Zwischenergebnis für 4 * fakultät(3): 24
Zwischenergebnis für 5 * fakultät(4): 120
120
```

5.9 Anonyme Funktionen

Bisher musste jeder Funktion über das `def`-Schlüsselwort explizit ein Name gegeben werden. Doch an vielen Stellen braucht man nur einmal eine einfache Funktionalität, und es wäre unnötiger Aufwand, dafür immer extra eine vollwertige Funktion zu definieren.

Ein Beispiel für eine solche Situation ist die integrierte Funktion `sorted`, die es erlaubt, als optionales Argument mit Namen `key=` eine Funktion anzugeben, die auf jedem Element des zu sortierenden Körpers vor der Vergleichsoperation ausgeführt wird.

Damit lässt sich zum Beispiel die Reihenfolge der Sortierung umdrehen:

Beispiel: Umgedrehte Sortierreihenfolge mit benannter Funktion

```
def negativ(x):
    return -x

# Test
>>> sorted([4,7,2,9])
[2,4,7,9]
>>> sorted([4,7,2,9], key=negativ)
[9,7,4,2]
```

Funktionen wie `negativ` sind in der Regel einfach, werden einmal benutzt und dann wieder vergessen.

Mithilfe des Schlüsselwortes `lambda` kann stattdessen eine kleine anonyme Funktion erstellt werden:

```
lambda x: -x
```

Auf das `lambda` folgen eine Parameterliste und ein Doppelpunkt. Hinter dem Doppelpunkt muss ein beliebiger logischer oder arithmetischer Ausdruck stehen, dessen Ergebnis als Rückgabewert verwendet wird. Da die lambda-Funktion kein `return`-Statement beinhaltet darf sich hinter dem Doppelpunkt nur genau ein (beliebig komplexer!) Ausdruck befinden. Komplexe Statements, wie z.B. Kontrollstrukturen, können nicht verwendet werden.

```
lambda a,b: 2*a + b
```

Ein Lambda-Ausdruck ergibt ein anonymes Funktionsobjekt und kann deswegen an jeder Stelle verwendet werden, wo eine Funktion erforderlich ist: Zum Beispiel in obigem `sorted`-Beispiel.

Beispiel: Umgedrehte Sortierreihenfolge mit anonymer Funktion

```
>>> sorted([4,7,2,9])
[2,4,7,9]
>>> sorted([4,7,2,9], key=lambda x: -x)
[9,7,4,2]
```

5.10 Sortieren von Datenstrukturen

Sehr häufig wird man Listen oder Dictionaries in einer bestimmten Ordnung anzeigen lassen wollen, zum Beispiel, um die größten Zahlen einer Liste oder die häufigsten Wörter einer Frequenzliste zu erfahren.

5.10.1 Sort oder Sorted?

Python 3 bietet zwei Funktionen zum Sortieren von Listen: `sorted(..)` und `list.sort()`.

`.sort()` ist eine Funktion des Datentyps `list`, und muss deswegen mit dem Punkt-Operator direkt auf einer Liste aufgerufen werden:

```
>>> zahlen = [7,4,2,9]
>>> print(zahlen)
[7,4,2,9]
>>> zahlen.sort()
>>> print(zahlen)
[2,4,7,9]
```

Achtung: `.sort()` hat keinen Rückgabewert, es kann also das Ergebnis **nicht** einer neuen Variablen zugewiesen werden. Stattdessen wird das Listen-Objekt, auf dem diese Funktion aufgerufen wird, selbst verändert.

`sorted(..)` arbeitet genau andersherum: Die als Argument angegebene Liste wird **nicht** verändert, stattdessen wird eine sortierte Kopie der Liste als Rückgabewert zurückgegeben. Deswegen: Wird dieser Rückgabewert nicht weiterverwendet, so bleibt der Aufruf von `sorted(..)` ohne Folgen. Dies soll in folgendem Beispiel deutlich werden:

```
>>> zahlen = [7,4,2,9]
>>> print(zahlen)
[7,4,2,9]
>>> sorted(zahlen)
>>> print(zahlen)
[7,4,2,9]   # Die Liste in der Variable zahlen wurde nicht verändert.
>>> neue_liste = sorted(liste)
>>> print(neue_liste)
[2,4,7,9]
>>> print(liste)
[7,4,2,9]
```

Ein weiterer Unterschied zwischen den beiden Funktionen ist, dass `.sort()` **nur** auf Objekten vom Datentypen Liste angewendet werden kann. `sorted(..)` hingegen kann beliebige sequenzielle Datentypen als Argument bekommen, also auch Dictionaries.

```
>>> lex = {"ü": 2, "z": 6,"a": 12}
>>> print(sorted(lex))
['a','z','ü']
```

Wie an diesem Beispiel erkennbar ist, gibt `sorted(..)` immer eine geordnete Liste zurück.

5.10.2 Reverse und die key-Funktion

Die folgenden Abschnitte sind sowohl für `.sort()` als auch für `sorted(..)` gültig.

Die Sortierfunktionen stellen uns zwei benannte Default-Argumente zur Verfügung, mit denen der Programmierer steuern kann, wie genau das Datenset sortiert werden soll.

Das erste dieser Default-Argumente ist das *reverse=False*-Argument, mit dem sich bestimmen lässt, ob die Sortierreihenfolge umgedreht werden soll. Das *reverse=*-Argument erwartet einen Wahrheitswert vom Typen `bool`

```
>>> zahlen = [7,4,2,9]
>>> print(sorted(zahlen))
[2,4,7,9]
>>> print(sorted(zahlen, reverse=True))
[9,7,4,2]
>>> print(sorted(zahlen, reverse=False))
[2,4,7,9]
```

Das zweite solcher Default-Argumente ist das *key=none*-Argument, welches es erlaubt eine Funktion anzugeben, die auf jedes zu vergleichende Element angewendet wird, bevor der zur Sortierung notwendige Vergleich durchgeführt wird.

Ein einfaches Beispiel hierfür ist das Sortieren von Worten, ohne Groß- und Kleinschreibung zu beachten:

```
>>> worte = ["aaa", "AAA", "bbb", "BBB"]
>>> print(sorted(worte))
['AAA','BBB','aaa','bbb']
>>> print(sorted(worte, key=str.lower)) # Funktion .lower() des Typen String
['aaa','AAA','bbb','BBB']
```

Mittels der im letzten Kapitel vorgestellten anonymen Funktionen lassen sich beliebige Sortieranweisungen kreieren:

```
>>> zahlen = [10,2,4,3,6,8,7,5,1,9]
>>> print(sorted(zahlen, key=lambda x: -x))
[10,9,8,7,6,5,4,3,2,1]
>>> print(sorted(zahlen, key=lambda x: x % 4))
[4,8,1,5,9,2,6,10,3,7]
```

5.10.3 Sortieren von Zeichenketten

Für Computerlinguisten interessanter ist aber die Sortierung von Datenstrukturen, die Zeichenketten, also Strings, beinhalten. Es wurde bereits festgestellt, dass die natürliche Ordnung der Buchstaben nach ihrem Encoding-Wert kein optimales Ergebnis liefert:

```
>>> worte = ["aaa", "AAA", "bbb", "BBB"]
>>> print(sorted(worte))
['AAA','BBB','aaa','bbb']
```

Das ignorieren von Groß- und Kleinschreibung hat dieses Problem schon deutlich verbessert:

```
>>> worte = ["aaa", "AAA", "bbb", "BBB"]
>>> print(sorted(worte))
['AAA','BBB','aaa','bbb']
>>> print(sorted(worte, key=str.lower)) # Funktion .lower() des Typen String
['aaa','AAA','bbb','BBB']
```

Kommen aber Zeichen ins Spiel, die auf der ASCII, Iso-Latin, und Unicode-Tabelle separiert sind, so stößt auch diese Methode an ihre Grenzen:

```
>>> wortliste = ["aaa", "xxx", "ZZZ", "www", "äää", "ÄÄÄ"]
# Natürliche Sortierung: Großbuchstaben vor Kleinbuchstaben
>>> sorted(wortliste)
['ZZZ', 'aaa', 'www', 'xxx', 'ÄÄÄ', 'äää']
# Sortierung von Kleinbuchstaben: Umlaute am Ende
>>> sorted(wortliste, key= str.lower)
['aaa', 'www', 'xxx', 'ZZZ', 'äää', 'ÄÄÄ']
```

Diesem Problem Zugrunde liegt die Frage, wie Buchstaben überhaupt sortiert werden sollen. An diesem Punkt scheiden sich die Geister: Wie soll zum Beispiel das ‘ü’ einsortiert werden? ‘t,u,ü,v,...,T,U,Ü,V’? Oder doch besser ‘t,u,v,w,x,y,z,ä,ö,ü,...,U,V,W,X,Y,Z,Ä,Ö,Ü’? Diese Problematik lässt sich nicht mehr mit einer einfachen anonymen oder benannten Funktion klären.

5.10.4 Sortieren mit locale

Glücklicherweise gibt es das zu Python gehörige `locale`-Modul, dass sich mit genau dieser Thematik befasst: Es stellt Funktionen und Klassen zur Verfügung, um mit Text lokalbezogen zu arbeiten.

Wie jedes Modul muss es über die import-Anweisung eingebunden werden, woraufhin es im Code verwendet werden kann:

```
import locale
```

Die wohl wichtigste Funktionalität, die das `locale`-Modul zur Verfügung stellt, ist es, die aktuelle Sprachumgebung einzustellen.
Die Funktionen des Moduls werden daraufhin die eingestellte Sprachumgebung beachten und entsprechend funktionieren.

Die Locale/Sprachumgebung wird folgendermaßen eingestellt:

```
>>> import locale

>>> locale.setlocale(locale.LC_ALL, "de_DE.UTF-8") # Programmweit; deutsch, UTF8.
```

Die Funktion `.setlocale(..)` wird auf dem locale-Objekt mittels des Punkt-Operators aufgerufen, und erwartet zwei Argumente:

Das erste Argument beschreibt die Reichweite der Einstellung.
Mögliche Werte sind: *LC_ALL, LC_CTYPE, LC_TIME, LC_NUMERIC*, und weitere.

Das zweite Argument ist ein Locale-String, der die zu verwendende Sprachumgebung angibt. Diese Strings müssen formatiert sein nach
ländercode.Sprachfamilie, also z.B. `"en_US.ASCII"`.

Ist das Locale gesetzt, kann endlich so sortiert werden, wie man es in einem echten Lexikon erwarten würde. Um dies zu ermöglichen, stellt das `locale`-Modul eine spezielle Funktion zur Verfügung, die das eingestellte Locale auf die Vergleichsoperation von `.sort()` oder `sorted(..)` anwendet.

Die Funktion heißt `locale.strxfrm`

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
>>> wortliste = ["aaa", "xxx", "ZZZ", "www", "äää", "ÄÄÄ"]
# Nat. Sortierung
>>> sorted(wortliste)
['ZZZ', 'aaa', 'www', 'xxx', 'ÄÄÄ', 'äää']
# Lowercase-Sortierung
>>> sorted(wortliste, key= str.lower)
['aaa', 'www', 'xxx', 'ZZZ', 'äää', 'ÄÄÄ']
# Locale-Sortierung
>>> sorted(wortliste, key= locale.strxfrm)
['aaa', 'äää', 'ÄÄÄ', 'www', 'xxx', 'ZZZ']
```

6 Linux

6.1 Einführung in Linux

6.1.1 Einleitung

Nach Bearbeitung dieses Kapitels können Sie grundlegende Eigenschaften von Linux Systemen verstehen.

6.1.2 Eigenschaften

- Multi-User-Betrieb
  - Am System können mehrere Benutzer gleichzeitig arbeiten. Sie werden über den Benutzernamen unterschieden.
- Multi-Tasking
  - Am System können mehrere Programme gleichzeitig arbeiten.
- Timesharing
  - Programme werden über Prioritäten verwaltet und bekommen abwechselnd die CPU(s).
- Interaktivität
  - Jeder Benutzer kann mit einem Terminal interaktiv arbeiten.
- Hierarchisches Dateisystem
  - Alle Dateien werden über eine Baumstruktur verwaltet.
- Benutzerschnittstelle (Shell)
  - Der Benutzer kommuniziert mit Linux über eine Kommandosprache, die Shell heißt.
- grafische Benutzerschnittstelle(GUI)
  - z.B. via X11-Server, KDE und Gnome.
- Programmentwicklung
  - Unter Linux gibt es zahlreiche Werkzeuge zur Programmerstellung.
- Textverarbeitung
  - Unter Linux gibt es zahlreiche Programme zur Textverwaltung.
- Netzwerkunterstützung
  - Jedes Linux kann in einem TCP/IP Netzwerk arbeiten. Unterstützt netzwerkweite Dateisysteme (NFS), mail, Programme usw.
- Kernel

  - Kontrolliert die Hardware
  - Memory Management Unit
  - CPU Vergabe über Prioritäten
  - Geräte Ein/Ausgabe
- System Dienste
  - Arbeiten direkt mit dem Kernel zusammen
  - Die Systemdienste stehen als C-Programme zur Verfügung.
- Dienstprogramme
  - Komplexe Programme als Linux Befehle
- Benutzerschnittstelle
  - Es stehen verschiedene Shells zur Verfügung :
  - bash, zsh
- Grafische Benutzerschnittstelle
  - basierend auf X11: KDE, Gnome und andere

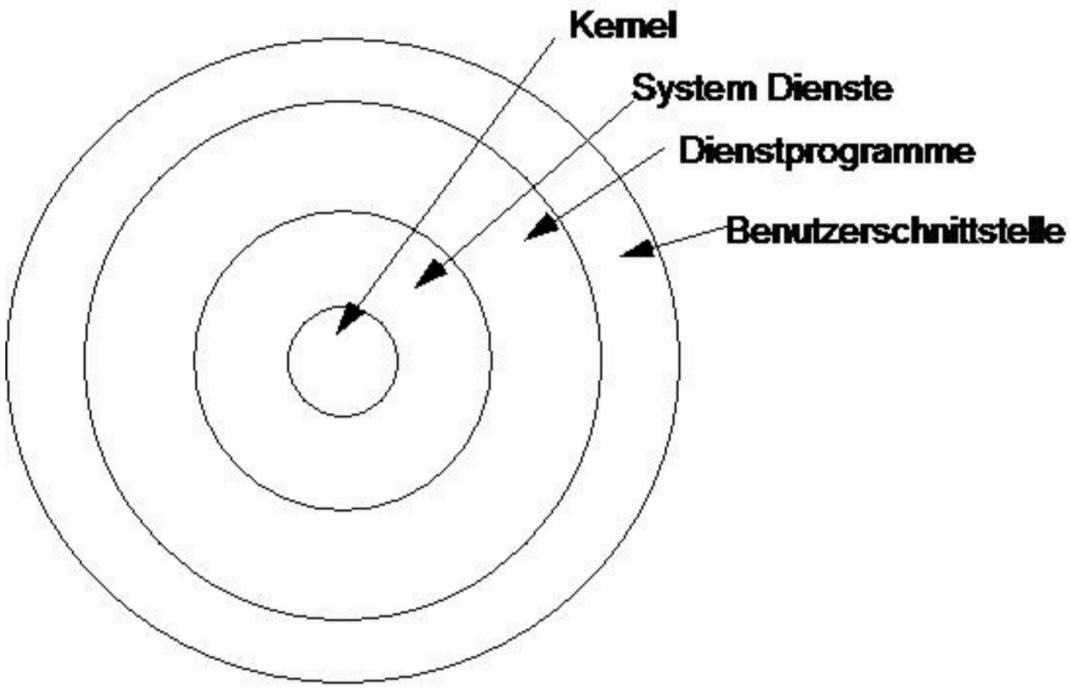


Abbildung 6.1: Graphische Darstellung des LINUX-Systems

6.2 Verwalten von Dateien Teil 1

6.2.1 Einleitung

Nach Bearbeitung dieses Kapitels können Sie :

- Das Linux Dateisystem verstehen.
- Verzeichnisse, Dateien und Gerätedateien unterscheiden.
- Die wichtigsten Aufgaben zum Verwalten von Dateien durchführen.
- Dateibezeichnungen korrekt eingeben.

6.2.2 Das Linux Dateisystem

Der hierarchische Dateibaum

Alle Linux Dateien sind streng hierarchisch baumartig organisiert. Der Ausgangspunkt des Dateibaums ist das `root` Directory `/`, auch Stammverzeichnis genannt, das auf weitere Dateien oder Verzeichnisse verweist. Es gibt keine Laufwerksbuchstaben.

Bei Linux gibt es Standardverzeichnisse in denen Dateien für spezielle Aufgaben zusammengefasst sind. Wichtige Standardverzeichnisse und die Aufgaben ihrer Dateien sind:

```
Verzeichnis         kurze Beschreibung
/                   root-Verzeichnis (Basisverzeichnis)
/sbin/              Programme für den Systemverwalter
/bin/               allgemeine Kommandozeilen-Befehle (GNU-Programme)
/boot/              Verzeichnis für Startdateien und den Kernel
/dev/               Gerätedateien (für Tastatur, Maus, Modem,
                    Festplatte, CDROM, ...)
/home/              Benutzerverzeichnisse
/root/              Verzeichnis des Systemverwalters
/tmp/               temporäre Dateien
/usr/               Dateien/Programme für die Benutzer
/usr/bin/           Anwendungsprogramme für die Benutzer
/usr/X11R6/bin/     Anwendungsprogramme für die grafische Oberfläche
/usr/src/           Quellcodes (z. B. vom Linux-Kernel)
/var/               Verzeichnis für sich ändernde Daten
                    (Mailverzeichnis, Druckerspooler, ...)
/var/log/           Protokoll-Dateien
/proc/              Systeminformationen; Schnittstelle zum Kernel
/opt/               optionale Programme (teilweise KDE, Firefox)
/mnt/               Mount-Point für Disketten, CD-ROMs, etc...
/media              Mount-Point für externe Festplatten, USB-Sticks,
                    CD-ROMs usw.
```

```
/etc/               systemweite Konfigurationsdateien
/etc/init.d/        Start-/Stopscripte der Systemdienste
/etc/X11/           Konfigurationsdateien von xorg
/lib/               allgemeine Programm-Bibliotheken (Programmlader,
                    Hauptbibliothek glibc)
/usr/lib/           Programm-Bibliotheken für Anwendungsprogramme
```

Jeder User arbeitet zu jedem Zeitpunkt in einem `Working Directory`, das er mit dem Befehl `cd` (change Directory) ändern und mit dem Befehl `pwd` (print Working Directory) ausgeben kann. Unter Linux können Verzeichnisse deviceübergreifend auf beliebigen Datenträgern abgespeichert werden. Der Systemadministrator bindet die Verzeichnisse an eine beliebige Stelle des Dateibaumes ein (mount), wobei sich an der hierarchischen Dateistruktur nichts ändert. Der User merkt auch nicht auf welcher physikalischen Platte er momentan arbeitet. (Abfrage mit `df .`) Eine aktuelle Zusammenfassung des hierarchischen Dateisystems findet sich unter `https://de.wikipedia.org/wiki/Filesystem_Hierarchy_Standard`.

Verzeichnisse, Dateien und Gerätedateien

In Linux gibt es

- gewöhnliche Dateien
  - Textdateien, am Terminal ausgebbar
  - Binärdateien, Programmdateien
- Verzeichnisse
  - Sammlung von anderen Dateien, die selbst Verzeichnisse sein können
- spezielle Dateien

Gerätedateien, die die Ein- und Ausgabe auf physikalische Geräte realisieren. Jedes Gerät hat für jede Schnittstelle ein eigenes Devicefile.

Wichtige Gerätedateien sind :

```
/dev/sd*       : Festplatten, allgemein Datenträger,
                 also auch usb-memory-sticks
/dev/hd*       : Festplatten, mit alten Treiber/p-ata
/dev/console   : System Konsole
/dev/tty01     : asynchrones Terminal
/dev/null      : Null-Device
/dev/sr*       : CD-Rom
/dev/zero      : Zero-Device
```

6.2.3 Fallbeispiel zum Verwalten von Dateien

Das Anlegen von Dateien

In den nächsten Kapiteln werden Datei- und Verzeichnisoperationen anhand eines Fallbeispiels erklärt. Es gibt 3 Studenten: meier, huber und schoch

Es ergibt sich folgende hierarchische Dateistruktur.

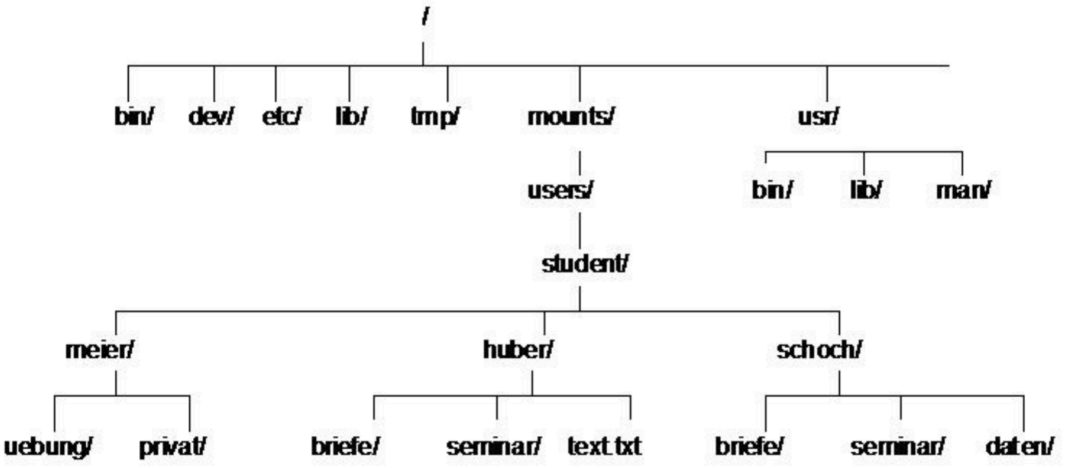


Abbildung 6.2: Hierarchische Dateistruktur von LINUX

Zum Anlegen von Dateien gibt es unter Linux die Möglichkeit, die Terminaleingabe in eine Datei umzuleiten (Befehl `cat`) oder einen Texteditor aufzurufen. Achtung: Unter Linux werden ältere Versionen von Dateien nicht automatisch gesichert, sondern stets überschrieben.

```
$ pwd
/mounts/users/student/meier
```

Im Bereich Holzbau soll für den Dachbau eine neue Datei dach angelegt werden:

```
$ cat > text1.txt
# Schreibt Terminaleingabe in die Datei text1.txt,
# Beenden der Eingabe mit CTRL d

$ touch text2.txt
# Kreiert eine leere Datei mit dem Namen text2.txt
```

Das Anzeigen von Inhaltsverzeichnissen

Der Inhalt des Verzeichnisses wird mit dem Befehl `ls` ausgegeben:

Wichtige Optionen des Befehls `ls`:

- `-a` listet auch die verborgenen Dateien (der Dateiname beginnt mit einem Punkt) im Verzeichnis auf.
- `-l` langes (ausführliches) Ausgabeformat
- `-F` hängt dem Dateiname ein Metazeichen an, das den Dateityp anzeigt:
  - `/` für Verzeichnis
  - `*` für ausführbar

In jedem Verzeichnis gibt es immer die zwei verborgenen Dateien:

- `.` Zeigt auf das eigene/aktuelle Verzeichnis
- `..` Zeigt auf das Elternverzeichnis des aktuellen Verzeichnisses

Beispiel: Benutzer huber nach dem einloggen:

```
$ pwd
/mounts/users/student/huber

$ ls
briefe seminar text.txt

$ ls -aF
./ ../ briefe/ seminar/ text.txt
```

Länge und Zeichenkonventionen von Dateinamen

Bei Datei- und Verzeichnisnamen können mit 254 Bytes nicht unbedingt 254 Stellen genutzt werden. Sonderzeichen und Umlaute sind nach Möglichkeit zu vermeiden aber nicht ausgeschlossen (Dateiname wird utf-8 codiert). Groß- und Kleinschreibung wird unterschieden(case sensitiv). Ein Name sollte nicht mit einem Punkt (nur bei explixiter verborgener Datei) oder mit einem Minus beginnen (Name könnte als Befehlsoption behandelt werden). Im Dateinamen sollte kein `*` stehen.

Wenn sich Sonderzeichen nicht vermeiden lassen, müssen sie mit besonderer Sorgfalt eingegeben werden:

z.B. `Datei\ mit \ Leerzeichen`

Der Schrägstrich `/` ist verboten, da er als Trenner im Pfadnamen fungiert.

Das Löschen von Dateien

Unter Linux können Dateien (Befehl `rm`) und Verzeichnisse mit ihren Unterverzeichnissen (Befehl `rm -r`) gelöscht werden.

Beispiel: Es sollen die Datei text.txt und das Verzeichnis `seminar` mit allen Dateien gelöscht werden:

```
$ pwd
/mounts/users/student/huber
$ rm text.txt
$ rm -r seminar
```

Um Dateien zu entfernen, deren Namen mit `-` beginnen, z.B. `-foo`, verwenden Sie eine der folgenden Anweisungen:

```
$ rm -- -foo
$ rm ./-foo
```

Das Anzeigen von Dateien

eine Textdatei kann am Bildschirm mit automatischem Scrolling (Befehl`cat`) oder bildschirmweise (Befehl `less`, `more`) ausgegeben werden :

```
$ pwd
/mounts/users/student/huber
```

Ausgabe der Datei `text.txt` am Bildschirm

automatisches Scrolling:

```
$ cat text.txt
```

bildschirmweise:

```
$ more text.txt
```

bildschirmweise, mit verbesserter Positioniermöglichkeit als `more`:

```
$ less text.txt
```

Das Verwalten von Verzeichnissen

Innerhalb von Verzeichnissen können beliebige neue Verzeichnisse kreiert (Befehl `mkdir`) oder, falls sie leer sind, gelöscht (Befehl `rmdir`) werden. Der Benutzer kann jedes Verzeichnis zu seinem Working Directory machen (Befehl `cd`).

Beispiel: das momentane Verzeichnis ist :

```
$ pwd
/mounts/users/student/huber
```

Ein Verzeichnis loesungen soll kreiert werden:

```
$ mkdir loesungen
```

Das Working Directory soll auf loesungen gesetzt werden:

```
$ cd loesungen
```

Der Benutzer will in sein Homeverzeichnis springen:

```
$ cd
```

Relative und absolute Datei- oder Verzeichnisbezeichnungen

Linux kann Dateien über relative oder absolute Dateibezeichnungen ansprechen. Während absolute Dateinamen die Dateihierarchie, ausgehend vom Rootdirectory im Namen auflisten, beginnt die Auflistung der Unterverzeichnisse bei relativen Bezeichnungen erst ab dem momentanen Working Directory.

```
$ pwd
/mounts/users/student/huber
```

Absolute Dateibezeichnung:

```
$ ls /mounts/users/student/huber/briefe
```

Relative Dateibezeichnung:

```
$ ls briefe
```

Das Kopieren und Umbenennen von Dateien

Unter UNIX kann eine Datei in eine andere kopiert werden (Befehl `cp`). Dabei werden bestehende Dateien überschrieben. Mehrere Dateien können mit einem `cp` Befehl nur in Verzeichnisse kopiert werden. Sollen die Namen von Dateien oder Verzeichnissen geändert werden, so gibt es unter UNIX den Befehl `mv` (=move).

```
$ pwd
/mounts/users/student/huber
```

Alle Dateien und alle Unterverzeichnisse aus dem Verzeichnis seminar/ sollen in das Verzeichnis seminar_alt/ kopiert werden:

```
$ cp -r seminar seminar_alt
```

Alle Dateien aber keine Unterverzeichnisse aus dem Verzeichnis `seminar/` sollen in das Verzeichnis `letzte/` kopiert werden:

```
$ mkdir letzte
$ cd seminar
$ cp * ../letzte
```

Alle Dateien und alle Unterverzeichnisse aus dem Verzeichnis letzte/ im Verzeichnis seminar/ sollen in das neue Verzeichnis /sicherung kopiert werden:

```
$ mkdir ../sicherung
$ cp -r ../letzte/seminar  ../sicherung
```

Das Verzeichnis letzte soll in `aller_letzte` umbenannt werden:

```
$ mv letzte aller_letzte
```

6.2.4 Dateinamen und der Einsatz von Wildcards

Einsatz von Wildcards

Um in einem Befehl mehrere Verzeichnisse oder Dateien anzusprechen, können in den Namen Wildcards verwendet werden.

Wichtige Wildcards sind `*` (steht für alles, außer einem Punkt am Anfang der Datei) und `?` (steht für ein Zeichen, i.d.R. einen Buchstaben/eine Ziffer)

```
$ pwd
/mounts/users/student/huber
```

Alle Dateien in allen Verzeichnissen, in deren Namen einf vorkommt, sollen aufgelistet werden:

```
$ ls */*/*einf*
```

Alle Dateien, die einen sechs Buchstaben langen Namen haben, sollen aufgelistet werden:

```
$ ls */*/??????
```

Verwendung von Sonderzeichen

Bestimmte Sonderzeichen in Dateinamen werden von der Shell interpretiert und zur Dateinamengenerierung verwendet.

Tabelle 6.1: Sonderzeichen

Sonderzeichen	Bedeutung
`\c`	Aufheben der Dateinamen Generierung für den nachfolgenden Buchstaben (hier `c`)
`\`	am Ende einer Zeile fordert eine Folgezeile
`" "`	Aufheben der Dateinamenexpandierung für den eingeschlossenen String
`~`	Die Shell (bash) ersetzt die tilde durch das Homedirectory

Ein Dateinname enthält ein Leerzeichen : Kreieren der Datei Mathe V1

```
$ cat > "Mathe V1"
```

Kreieren und Ansprechen der Datei Mathe

```
$ cat > Mathe\*
$ ls Mathe\*
```

6.2.5 Das Wichtigste in Kürze

- Das Linux Dateisystem ist ein hierarchisches, baumartiges Dateisystem, bei dem Verzeichnisse Device-übergreifend abgespeichert sein können. Das oberste Verzeichnis heißt root Directory.
- In Linux gibt es gewöhnliche Dateien, Verzeichnisse (Directories) und spezielle Dateien (Device Files). Je nach Aufgaben sind sie in Standardverzeichnissen abgespeichert.
- Datei und Verzeichnisnamen dürfen 254 Bytes lang sein, Groß- und Kleinbuchstaben werden unterschieden, Wildcards (`*`, `?`, `\`, `~`) werden unterstützt.
- Zu jedem Zeitpunkt befindet sich der User in einem Working Directory, das er mit `pwd` ausgeben und mit `cd` ändern kann.
  - Datei- und Verzeichnisbezeichnungen können absolut (beginnend beim root Directory) und relativ (beginnend beim Working Directory) angegeben werden.
  - Der User kann Dateien anlegen (Befehl `cat`), am Terminal anzeigen (Befehl `cat`), kopieren (Befehl `cp`), umbenennen (Befehl `mv`) und löschen (Befehl `rm`).
  - Mehrere Dateien können nur in ein Verzeichnis kopiert werden. Verzeichnisse können neu angelegt (Befehl `mkdir`) und falls falls sie leer sind, können Sie mit dem Befehl `rmdir`, sonst mit dem Befehl `rm -r` gelöscht werden.

  - Beim Kopieren und Umbenennen von Dateien verwaltet Linux keine alten Versionen von Dateien, sondern überschreibt sie.

6.2.6 Tipps für die Praxis

- **Gemeinsame Optionen `-i`, `-r`, `-f` bei Dateioperationen**
  - Bei den Befehlen `rm`, `cp` und `mv` haben die Optionen `-i`, `-r` und `-f` die gleiche Funktionalität:
    * `-i` (interactive), fragt den Benutzer bevor eine Datei überschrieben wird.
    * `-r` (recursiv) Bei der Dateinamengenerierung werden auch Dateinamen aus den Unterdirectories generiert.
    * `-f` (force) Keine Nachfragen.
- **Wildcard `*` und die Option `-R` beim `ls` Befehl**

Wird `ls *` aufgerufen, so werden auch die Inhalte aller Unterverzeichnisse ausgegeben. `ls -d *` listet nur die Namen der Verzeichnisse auf.

Die Option `-R` listet den Inhalt und den Namen aller Verzeichnisse und Unterverzeichnisse auf.

- **More Befehl in Pipes**

Erzeugt ein Befehl eine Terminalausgabe, die länger als eine Bildschirmseite ist, so kann die Ausgabe, auf den less Befehl "gepiped", also bildschirmweise ausgegeben werden. `ls * | less`

- **Automatische Dateinamengenerierung bei Kommanodaufrufen.**

Bei allen Kommandoaufrufen ist die automatische Dateinamengenerierung der Shell eingeschaltet:

`cd semin*` ist äquivalent zu `cd seminar`.

Die Kommandozeile kann auch durch die Tabulatortaste automatisch vervollständigt werden.

`cd sem <tab>` wird zu `cd seminar`, falls es nur ein Verzeichnis mit sem beginnend gibt, anderenfalls erscheint nur ein Teilstück bzw. beim zweiten Drücken von `<tab>` eine Auswahl.

- **Verhindern des automatischen Löschens oder Überschreibens von Dateien**

Da Linux alte Versionen von Dateien bei unveränderten Voreinstellungen überschreibt, empfiehlt es sich für die Befehl `mv, cp` und `rm` alias Definitionen mit der interactive Option einzuführen. Die alias Definitionen werden mit der Option `-f` überschrieben.

```
alias rm='rm -i'
alias cp='cp -i'
```

```
alias mv='mv -i'
```

Die alias-Funktion kann durch Voranstellen eines `\` oder durch Angabe des vollständigen Pfades `/usr/bin/rm`, durch Hochkomma `'rm'` oder Anführungszeichen `"rm"` kurzzeitig umgangen werden.

- **Wildcards bei Verborgenen Dateien**

Wildcards in einem Dateinamen generieren keinen Dateinamen, der mit einem Punkt beginnt (=Verborgene Datei).

Abhilfe: `ls .*`

6.3 Editieren von Texten

6.3.1 Einleitung

Unter Linux gibt es mehrere Editoren mit verschiedenen Eigenschaften für unterschiedliche Einsatzgebiete :

- vi: Fullscreen Editor, umschaltbar in den ex-Editor-Modus
- emacs: wichtiger GNU GPL Editor unter LINUX mit Arbeitsumgebung
- kate: ist ein freier Texteditor für Programmierer.

Diese drei Editoren haben verschiedene technische Bedingungen für die Nutzung. Der vi läuft in jeder Konsole (Textmodus, xterm). Der emacs kann sowohl in der Konsole laufen mit der Option `-nw`, als auch im grafischen Modus. Kate dagegen läuft nur im Grafikmodus, genauer mit Hilfe von KDE- und QT-Bibliotheken.

Ein weiteres Beispiel für einen simplen Editor in der Textkonsole ist mcedit oder nano. Gedit ist ein weiterer Editor im Grafikmodus.

In diesem Kapitel lernen Sie:

- Aufrufen von kate
- Die Fenster von kate
- Die Menüeinträge von kate
- Erstellen einer neuen Datei
- Spezielle Kommandos zum praktischen Gebrauch

Aus dem Handbuch zu kate:

http://docs.kde.org/stable/de/kdesdk/kate/fundamentals.html#starting-kate

6.3.2 Aufrufen des kate Editors

kate wird vom K-Menü oder von der Befehlszeile aufgerufen.

Vom K-Menü

Das KDE → Programmmenü wird durch Klicken auf den großen K-Knopf oder das jeweilige Symbol der Linux Version links unten auf dem Bildschirm in der Werkzeugleiste geöffnet. Zeigen Sie mit der Maus auf den Menüpunkt Programme, Dienstprogramme, Editoren. Es erscheint eine Liste der verfügbaren Editoren. Wählen Sie kate (auch aufgeführt als erweiterter Texteditor). kate lädt die zuletzt bearbeiteten Dateien, außer Sie haben eingestellt, dass dies nicht der Fall sein soll. Sehen Sie unter "Einstellungen - kate einrichten" nach, wie Sie diese Funktion ein- und ausschalten können.

Von der Befehlszeile

Sie können kate auch von der Befehlszeile starten. Wenn Sie dabei gleichzeitig einen Dateinamen angeben, wird diese Datei durch kate geöffnet oder erzeugt.

Verschiedene Modi von kate:

Textdatei

Mit Hilfe von kate kann man eine Textdatei erstellen indem man beim Speichern oder Erstellen die Endung `.txt` hinzufügt.

.py Datei

Kate erkennt eine Pythondatei an der Endung `.py`

Durch den Header (auch shebang, magic line)

`#!/usr/bin/python` für Python

`#!/usr/bin/python3` für Pythn3

`#!/usr/bin/env python` für Python irgendwo im System

kann man die Dateien auch ohne Voranstellen von `python` ausführen

Sitzungen:

Sitzungen erlauben Ihnen mehrere Listen von Dokumenten und Einstellungen für das Benutzen in kate zu speichern. Sie können beliebig viele Sitzungen speichern, und Sie können unbenannte oder anonyme Sitzungen für das einmalige Bearbeiten von Dokumenten verwenden.

Neue Sitzung:

Wenn Sie eine neue Sitzung starten, dann wird das Dialogfenster für die Standardsitzung geöffnet. Zum Speichern der Fenstereinstellungen in der Standardsitzung müssen Sie das Feld "Fenstereinstellungen einbeziehen" auf der Karte "Sitzungsverwaltung" unter "Einstellungen - kate einrichten" einschalten, dann die Standardsitzung laden, die Fenstereinstellungen anpassen und die Sitzung speichern.

Letzte Sitzung:

Wenn sie auf "Sitzung öffnen" klicken so wird die alte von Ihnen verwendete Sitzung geöffnet mit allen Fenstern, Verzeichnissen und Dateien mit denen schon gearbeitet wurde. Im Editorbereich erscheint die letzte bearbeitete Datei.

Beispiel: Kate: Öffnen einer bestehenden Datei

```
kate Dateiname.txt
kate datei.py
```

Erstellen einer neuen Datei

Es gibt 2 Möglichkeiten mit Hilfe von kate eine Datei zu erstellen:

- In der Befehlszeile wird der Name einer neuen Datei angegeben:

  ```
  kate neuedatei.py
  ```

- Kate aufrufen: `kate`

  Datei → Neu

kate ünterstützt das Drag-and-Drop-Protokoll von KDE. Dateien können gezogen und auf kate abgelegt werden; von der Arbeitsoberfläche, Dolphin, oder einer FTP-Seite, die in einem Dolphin-Fenster geöffnet ist.

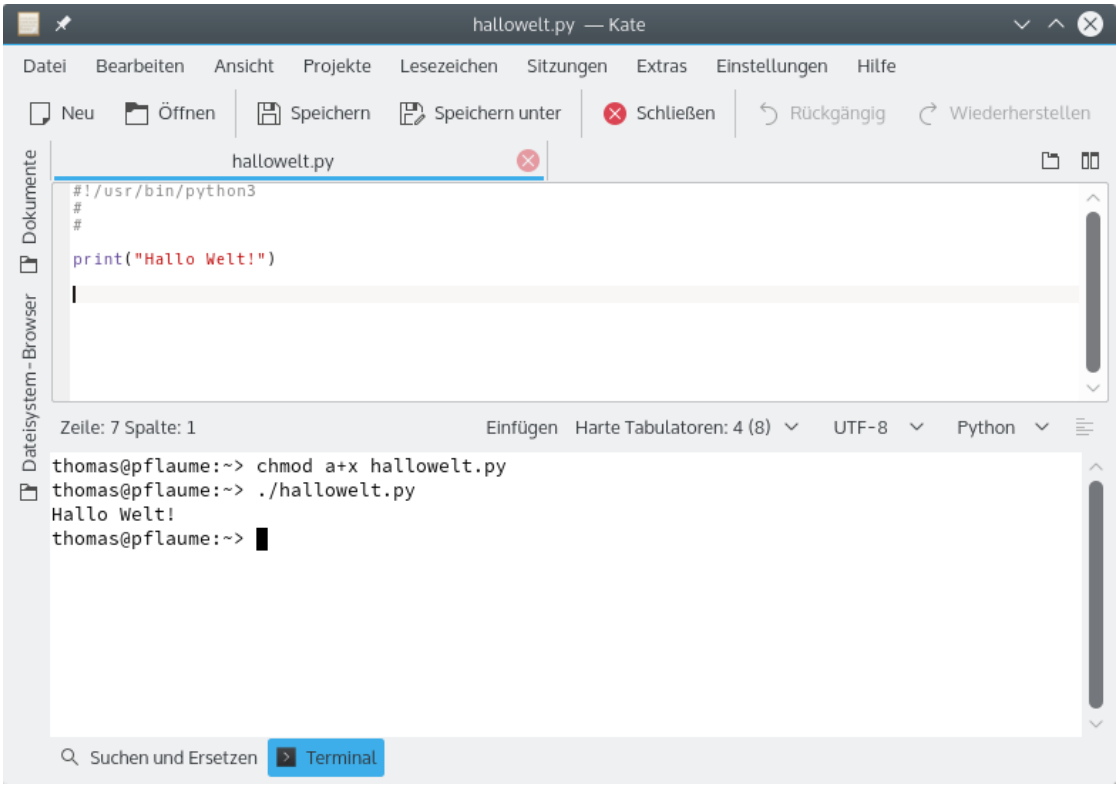


Abbildung 6.3: kate screenshot

Die Fenster des kate Editors

Kate ist in verschiedene Bereiche geteilt : Das **Hauptfenster** von kate ist ein Standard-KDE-Anwendungsfenster mit einer zusätzlichen Seitenleiste, die Werkzeugansichten enthält. Es hat eine Menüleiste mit den Standardmenüs und einigen mehr, sowie eine Werkzeugleiste mit Knöpfen für oft benutzte Befehle.

Der wichtigste Teil des Fensters ist der **Editorbereich**, der standardmäßig einen Texteditor anzeigt, in dem Sie Ihr Dokument bearbeiten können.

Die **Dokumentenliste** zeigt alle aktuell in Kate geöffneten Dateien an. Dateien, die noch nicht gesicherte Änderungen enthalten, werden mit einem kleinen DiskettenSymbol links neben dem Dateinamen gekennzeichnet.

Wenn zwei oder mehrere Dateien mit dem selben Namen (in verschiedenen Verzeichnissen) geöffnet sind, wird an die Namen eine Zahl angehängt z. B. `(2)` usw. Für die Auswahl der gewünschten Datei wird ihr Name einschließlich Pfad in der Kurzinfo angezeigt.

Wenn Sie ein Dokument aus der Liste aktiv machen wollen, klicken Sie einfach auf den Namen des Dokuments in der Liste.

Die Dokumentenliste stellt standardmäßig die Einträge farbig dar: Die Einträge der zuletzt bearbeiteten Dokumente werden mit einer Hintergrundfarbe hervorgehoben, Dokumente, die tatsächlich bearbeitet wurden, werden durch eine zusätzlich eingeblendete Farbe hervorgehoben. Das Dokument, das zuletzt bearbeitet wurde, hat die stärkste Farbe, so dass Sie die Dokumente, an denen Sie aktuell arbeiten, einfach finden können. Diese Hervorhebungen können Sie im Einrichtungsdialog für die Dokumentliste einrichten.

Der **Dateisystem-Browser** ist ein Verzeichnisanzeiger, von dem aus Sie Dateien im aktuell angezeigten Verzeichnis öffnen können.

Von oben nach unten besteht der Dateisystem-Browser aus folgenden Elementen:

Einer Werkzeugleiste, diese enthält Standardnavigationsknöpfe zum schnellen Wechsel der Verzeichnisse.

Die Standardposition der Dokumentliste sowie vom Dateisystem-Browser ist links vom Editorbereich im Kate-Fenster. Der eingebaute Terminal-Emulator ist eine Kopie der KDE-Anwendung Konsole, er ist durch Wählen von Fenster → Werkzeugansichten → Terminal anzuzeigen und bekommt beim Einschalten den Fokus. Wenn die Option “Automatisch mit dem aktuellen Dokument abgleichen” im Feld “Einstellungen” von Kate einrichten eingeschaltet ist, wird das Verzeichnis des Terminal-Emulators in das Herkunftsverzeichnis der aktuellen Datei umgeschaltet, wenn dies möglich ist.

Die Standardposition ist unten im Kate-Fenster, unterhalb des Editorfensters.

6.3.3 Wichtige Menüeinträge

Das Menü Datei

- Datei → Neu (Strg+N)
  - Dieser Befehl startet eine neue Datei im Editorfenster. In der Dateiliste links wird die neue Datei als "Unbenannt" bezeichnet.
- Datei → Öffnen (Strg+O)
  - Öffnet das Standard-KDE Dialogfenster, das das Öffnen einer oder mehrerer Dateien erlaubt.
- Datei → Zuletzt geöffnete Dateien
  - Dieser Befehl öffnet ein Untermenü mit einer Liste der zuletzt bearbeiteten Dateien. Sie können daraus direkt eine dieser Dateien öffnen.
- Datei → Öffnen mit
  - Dieses Untermenü enthält eine Liste mit Anwendungen, die den MIME-Typ des aktuellen Dokumentes verarbeiten können. Das Klicken auf einen Eintrag öffnet das aktuelle Dokument mit dieser Anwendung.
  - Weiterhin gibt es einen Eintrag "Sonstige". Dieser Befehl öffnet ein Dialogfenster, in dem Sie eine andere Anwendung auswählen können, in der die aktive Datei geöffnet werden soll. Die aktive Datei bleibt weiterhin in Kate geöffnet.
- Datei → Speichern (Strg+S)
  - Dieser Befehl speichert Ihre Datei. Benutzen Sie ihn oft. Solange die Datei noch den Namen Unbenannt trägt, wird automatisch anstelle von Speichern der Befehl "Speichern unter" ausgeführt.
- Datei → Speichern unter
  - Mit diesem Befehl benennen Sie Dateien oder benennen diese um. Es wird das Dialogfenster zum Speichern einer Datei aufgerufen. Dieses Dialogfenster funktioniert genauso wie das Dialogfenster zum Öffnen einer Datei. Sie können sich durch das Dateisystem bewegen, eine Vorschau von vorhandenen Dateien ansehen oder Masken zur Dateiauswahl benutzen.
  - Geben Sie den Namen, den Sie der Datei geben wollen, in das Feld ein und klicken Sie auf OK. Achten Sie dabei auch auf die Kodierung.

Das Menü Bearbeiten

- Bearbeiten → Überschreibmodus (Einfg)
  - Schaltet zwischen den beiden Arten des Eingabemodus um. Wenn der Modus EINF ist, dann setzen Sie die eingegebenen Zeichen an der Stelle des Cursors ein. Wenn der Modus Überschr. ist, dann ersetzt jedes eingegebene Zeichen ein Zeichen rechts vom Cursor. Die Statusleiste zeigt den aktuellen Status des Auswahlmodus an, entweder EINF oder Überschr.

Das Menü Ansicht

- Ansicht → Zeilennummern anzeigen (F11)
  - Mit diesem Eintrag wird ein zusätzlicher Rand an der linken Seite des aktiven Rahmens ein- oder ausgeschaltet, der Zeilennummern anzeigt.

Das Menü Extras

- Extras → Terminal mit aktuellem Dokument synchronisieren
  - Diese Option bewirkt, dass der eingebaute Terminal-Emulator immer mit cd zum Ordner des aktuellen Dokuments wechselt, wenn ein neues Dokument geöffnet wird oder zu einem anderen Dokument umgeschaltet wird.
- Extras → Kodierung
  - Sie können die Standardeinstellung für die Kodierung, die in Einstellungen → Kate einrichten … Öffnen/Speichern festgelegt ist, für dieses Dokument hier verändern.
- Extras → Kommentar
  - Dies fügt das Zeichen für eine Kommentarzeile und ein Leerzeichen an den Zeilenanfang der aktuellen Zeile oder der aktuellen Auswahl hinzu.
- Extras → Kommentar entfernen
  - Dies entfernt das Zeichen für eine Kommentarzeile und ein Leerzeichen (sofern vorhanden) am Zeilenanfang der aktuellen Zeile oder der aktuellen Auswahl.
- Extras → Großschreibung
  - Der ausgewählte Text oder der Buchstabe nach dem Cursor wird in Großschreibung gesetzt.
- Extras → Kleinschreibung
  - Der ausgewählte Text oder der Buchstabe nach dem Cursor wird in Kleinschreibung gesetzt.
- Extras → Großschreibung am Wortanfang
  - Setzt den ausgewählten Text oder das aktuelle Wort in Großbuchstaben.

6.3.4 Spezielle Kommandos zum praktischen Gebrauch

Strg+Z Abbrechen und Rückgängigmachen eines kate Kommandos.

Strg+S Speichern der aktuellen Datei im aktuellen Zustand.

Strg+Q Beenden - Aktives Editorfenster schließen.

6.3.5 Andere Editoren

SED: Der sed ist ein batchorientierter Editor. Die Editor Kommandos werden in der Aufrufzeile oder in Scriptfiles eingegeben. Der sed arbeitet die Befehle sequentiell ab und

schreibt das Ergebnis des Editierens auf das Terminal.

Aufruf:

```
sed [-n] -f scriptfile [file-list]
sed -n scriptfile [file-list]
```

```
sed s/abc/def/d  test
```

ersetzt den String “abc” durch den String “def” in jeder Zeile des Files “test”

```
sed -f script-file test
```

bearbeitet die Datei “test” mit den Kommandos aus “script-file”

GNU - EMACS: Er ist ein freier (GPL) fullscreen Editor, der für zahlreiche Linux Systeme implementiert ist. Er bietet eine komplexe Arbeitsumgebung zum gleichzeitigen Editieren, Programmerstellen und Arbeiten mit der Shell. Außerdem erlaubt er dem Benutzer, eine eigene Editorumgebung mit LISP ähnlichen Befehlmacros zu erstellen.

6.4 Verwalten von Dateien Teil 2

6.4.1 Die einzelnen Merkmale einer Datei

Die Dateimerkmale im Überblick

Der Befehl `ls -lisg` listet alle Dateimerkmale :

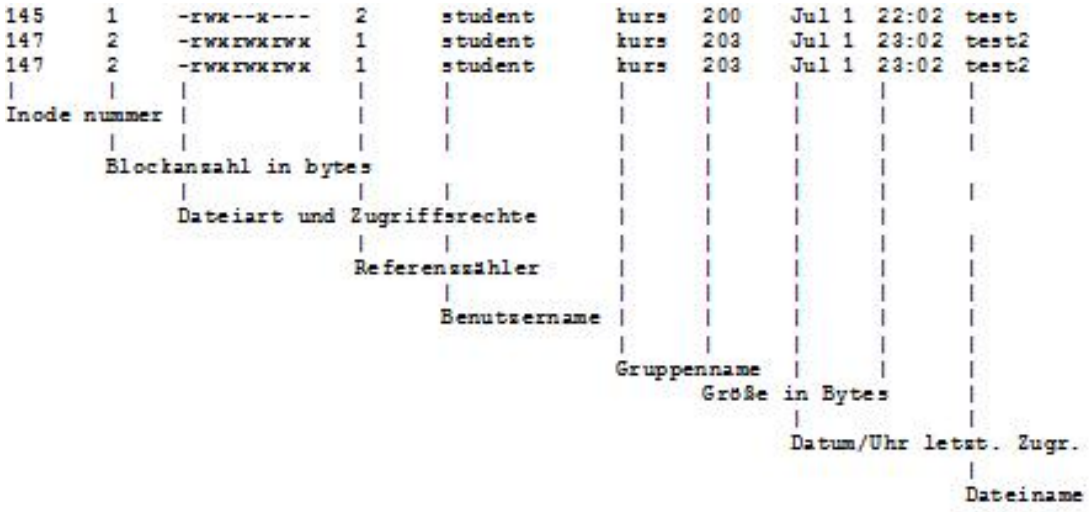


Abbildung 6.4: Dateimerkmale im Überblick

6.4.2 Gruppen und Owner

- Jedem Benutzer wird unter Linux in der Datei `/etc/passwd` eine eindeutige Benutzer- (Benutzer ID) und eine Gruppennummer (Gruppen ID) zugeordnet.

Der UNIX Befehl `id` gibt die Benutzer und Gruppeneinstellungen an.

```
$ id
```

- In einer Gruppe können mehrere Benutzer zusammengefasst werden.
- Unter Linux kann jeder Benutzer Mitglied mehrerer Gruppen sein.
- In der Datei `/etc/group` wird festgelegt, welcher Benutzer zu welcher Gruppe gehört. Außerdem wird in dieser Datei für den Gruppen ID ein symbolischer Name definiert.
- Der Loginname ist der symbolische Name für den Benutzer ID.
- Benutzer IDs liegen zwischen 0 und 65535, wobei 0 für den Superuser reserviert ist. Die Nummern 1-99 sind ebenfalls reserviert.

Zugriffsrechte unter Linux

Unter Linux werden folgende drei Klassen von Benutzern unterschieden :

u: user, gleiche User-ID

g: group, gleiche Gruppen-ID

o: beliebige User- und Gruppen-ID, ungleich der eigenen

Für jede dieser drei Benutzerklassen werden drei verschiedene Arten von Zugriff unterschieden:

r: Lesender Zugriff

w: Schreibender Zugriff

x: Ausführender Zugriff

Je nach Art der Datei haben die Zugriffsrechte verschiedene Bedeutungen:

Tabelle 6.2: Zugriffsrechte

Dateiart	read	write	execute
Gerätedatei	darf vom Gerät lesen	darf auf Gerät schreiben	
Verzeichnis	auf Dateien darf zugegriffen werden (Name)	Dateien im Verzeichnis dürfen gelöscht/angelegt werden	In das Verzeichnis darf mit `cd` gewechselt werden
normale Datei	lesen erlaubt	ändern erlaubt	ausführen erlaubt

Strategie zum Test der Zugriffsrechte bei Dateien

```
Benutzer ID = Benutzer ID der Datei
   |
   |-Ja-> Erlauben Zugriffsrechte für user Aktion -Ja-> Aktion ausführen
   |
   | Nein
Gruppen ID des Benutzers = Gruppen ID der Datei
   |
   |-Ja-> Erlauben Zugriffsrechte für group Aktion -Ja-> Aktion ausführen
   |
   |
```

```
    | Nein
  Falls UNIX = BSD-Typ erneuter Test:
    Nachprüfen in /etc/group: gehört der Benutzer zur Gruppe des Dateibesitzers
    |      |
    |      |-Ja-> Erlauben Zugriffsrechte für group Aktion -Ja-> Aktion ausführen
    |      |
    |      | Nein
    |      |
 Erlauben die Zugriffsrechte für other die Aktion --Ja--> Aktion ausführen
    |      |
    | Nein |
    |      |
    Aktion verbieten
```

Ändern der Zugriffsrechte

Die Zugriffsrechte einer Datei können mit symbolischen Werten oder mit einer Protectionmaske geändert werden.

Ändern mit symbolischen Werten :

```
chmod  u <op> rwx   dateiname oder Verzeichnisname
chmod  g <op> rwx   dateiname oder Verzeichnisname
chmod  o <op> rwx   dateiname oder Verzeichnisname
chmod  a <op> rwx   dateiname oder Verzeichnisname

u : user, g : group, o : other, a : all

<op> ... =  explizites Setzen des angegebenen Rechts
<op> ... -  entfernen des angegebenen Rechts
<op> ... +  hinzufügen des angegebenen Rechts
```

```
chmod a+r test
```

: alle erhalten das Recht zum Lesen

```
chmod ug+wx test
```

: User und Gruppe erhält das Recht zum Schreiben und Ausführen

Ändern mit einer Oktalzahl/Maske :

Der Befehl lautet:

```
chmod maske dateiname

user          |group        |other
--------------|-------------|------------
```

```
r    w    x   |r   w   x    |r   w   x
0/1  0/1  0/1 |0/1 0/1 0/1  |0/1 0/1 0/1
4    2    1   |4   2   1    |4   2   1
erste Ziffer  |zweite Ziffer|dritte Ziffer
```

Ein Bit mit “1” bedeutet, dass das Recht für die spezifische Gruppe gesetzt ist.

Es sollen folgende Rechte vergeben werden :

user = rw, group = x, other = -

daraus errechnet sich folgende Maske:

```
user  = rw     110 → 4 · 1 + 2 · 1 +1 · 0 = 6
group = x      001 → 4 · 0 + 2 · 0 +1 · 1 = 1
other = -      000 → 4 · 0 + 2 · 0 +1 · 0 = 0
                                    Maske = 610
```

Der Befehl lautet (ein Beispiel):

```
$ chmod 610 test
```

Einstellen der Defaultmaske

Wird eine Datei kreiert, so wird ihr Zugriffschutz nach einem Defaultwert festgelegt. Der Defaultwert kann mit dem Befehl `umask` gesetzt werden.

Ändern der Gruppenzugehörigkeit einer Datei

Der Besitzer einer Datei kann, falls er Mitglied dieser Gruppe ist, die Gruppenzugehörigkeit einer Datei ändern.

Der Befehl lautet (ein Beispiel):

```
$ chgrp student2 test
```

Ändern des Owners einer Datei

Nur der Superuser kann den Owner einer Datei ändern.

Der Befehl lautet (ein Beispiel) :

```
$ chown student test
```

6.4.3 Die Bedeutung von Devicegrenzen

Plattenplatzbelegung

Jede Plattenpartition entspricht im System einem Device.

Damit der Benutzer mit einem Device über eine Dateistruktur arbeiten kann, muss die Partition initialisiert werden, wodurch die Partition als Dateisystem ansprechbar wird.

Ein Benutzer kann erst dann mit einem Dateisystem arbeiten, wenn es mit dem `mount` Befehl in das hierarchische Dateisystem von Linux eingebunden ist. Beim `mount` Befehl muss der Benutzer das Verzeichnis angeben, in das er das Dateisystem einbindet.

Der Befehl lautet (ein Beispiel) :

```
$ mount /dev/sdc1 /usr
```

Nachdem das Dateisystem "gemounted" ist, sind seine Dateien über die hierarchische Dateistruktur zugänglich. Mit dem Befehl `cd` kann der Benutzer in jedes Verzeichnis springen.

Der Befehl `mount` ohne Argumente gibt an, welche Dateisysteme in welches Verzeichnis gemounted sind. Der Befehl `df` zeigt es etwas übersichtlicher gleich mit Belegung an:

```
$ df
Filesystem        Total    kbytes   kbytes   %
node             kbytes    used     free     used  Mounted on
/dev/sda1         15343    9765    4044      71%   /
/dev/sdb1        191958  163585    9178      95%   /usr
/dev/sdc1         49279   35474    8878      80%   /usr/users
```

Der Symbolic Link

Unter Linux kann man mit einem Dateinamen auf eine Datei zeigen, die auf einem anderen Device liegt.

Die Verbindung zwischen Dateien über die Devicegrenzen hinweg funktioniert nur mit einem `symbolic link`, innerhalb eines Devices sind auch `hard links` möglich.

Bei einem symbolic link wird im Header der Datei nicht die Inode - Nummer der Datei auf die er zeigt, sondern der komplette Dateinamen, zusammen mit seinem Pfad eingetragen.

Der Referenzzähler erhöht sich bei symbolic links nicht.

Die Datei benutzer soll auf das Verzeichnis `/usr/users` zeigen. Die Datei benutzer darf vorher nicht existieren:

```
$ ln -s /usr/users benutzer
```

```
$ ls -l benutzer
lrwxr-xr-x  1 kurs             5 Jul  4 15:30 benutzer -> /usr/users
```

6.4.4 Das Wichtigste in Kürze

Die wichtigsten Dateimerkmale einer Datei sind:

- Datei Eigentümer und - Gruppe
- Dateischutz und andere Dateistatistik
- Größe der Datei in Bytes
- Blockadresse der Daten der Datei auf der Platte
- Zeit der Datei - Erzeugung
- Zeit des letzten lesenden Zugriffs
- Zeit des letzten schreibenden Zugriffs
- Anzahl der Verweise auf die Datei
- Jedem Benutzer unter Linux wird über seinen Benutzernamen eine Benutzer-ID und eine Gruppen-ID zugeordnet.
- Der Benutzer root (Benutzer ID = 0) hat im Linuxsystem privilegierte Rechte und kann alle Datei- und Systemattribute ändern.
- Ein Benutzer kann Mitglied mehrerer Gruppen sein.
- Jede Datei gehört einem Benutzer. Über Zugriffsrechte wird geregelt, für was eine Datei verwendet werden kann.
- Bei den Zugriffsrechten werden drei Klassen unterschieden :`user`, `group` und `other`.
- Für jede Klasse kann man einen `read`, `write` oder `execute` Zugriff festlegen.
- Beim Bearbeiten einer Datei prüft Linux zu welcher Klasse der Benutzer relativ vom Besitzer der Datei gehört:
  - user : gleiche Benutzer ID
  - group : gleiche Gruppen ID
  - other : beliebige Benutzer- oder Gruppen-ID.
- Die Zugriffsrechte unter Linux können symbolisch oder mit Oktalzahl gesetzt oder geändert werden.
- Ein Datenträger wird unter Linux in Partitionen unterteilt, die, falls sie initialisiert sind, ein eigenständiges Dateisystem haben.
- Dateisysteme werden in die hierarchische Dateistruktur von Linux eingebunden (gemounted)

6.5 UNIX-Befehle

6.5.1 Hilfe (Befehlsoptionen, Parameter, generelle Verwendungsweise, Verwendungsbeispiele)

- `man` Befehl
- `info` Befehl

Aufbau von `man` und `info` pages (Auszug):

- NAME (Names des Programms)
- SYNOPSIS (generelle Struktur des Befehls)
- DESCRIPTION (kurze Zusammenfassung, was der Befehl tut)
- OPTIONS (Optionen, Parameter und deren Beschreibung)
- EXAMPLES (Beispiele zur Verwendung, empfehlenswert!)
- SEE ALSO (man pages von ähnlichen Programmen oder config files)

`-h` oder `--help`

Meist stehen auf beiden Seiten die gleichen Informationen, aber z.B. bei GNU Programmen wird man in den Info Pages eher fündig (manchmal auch umgekehrt).

6.5.2 Einsatz der UNIX-Shell: Pipes, $<$, $>$ (Ein- und Ausgabeumleitung)

```
$ sort liste > sortierte_liste
```

Hier wird die Ausgabe von `sort` in eine Datei umgeleitet.

```
$ ls -l | less
```

Umleitung der Ausgabe von `ls` in das Programm `less`. Mehrfachverkettungen mittels Pipe sind auch möglich s.u.

```
$ cat File
```

Schreibt ein File in die Standardausgabe (meist das aktuelle Terminal), von der aus es weiterverarbeitet werden kann.

```
$ echo String
```

Zeigt den String in STDOUT an

```
$ echo $Var
```

Zeigt den Inhalt der Variable Var in STDOUT an

```
$ tail -n
```

Zeigt die letzten n Zeilen der Datei an.

```
$ head -n
```

Zeigt die ersten n Zeilen der Datei an.

6.5.3 Kommandooptionen

Welche Kommandooptionen:

```
$ cmd --help

-s value     # kurze Variante
--size=value # lange Variante
```

6.5.4 Shells: bash, tcsh

```
piping |

redirecting > >> < << &1 &2
```

Control zur Steuerung von Prozessen:

```
^C Abbruch
^S Anhalten der Ausgabe aufs Terminal
^Q resume der Ausgabe aufs Terminal
^Z schicke Prozess in den Hintergrund:
     neue Befehle z.B. bg oder fg damit er weiter läuft,
     können eingegeben werden
```

6.5.5 Was ist los auf dem Rechner?

```
$ top
$ ps
```

listet eigene Prozesse auf Wichtig ist die LOAD des Rechners im Verhältnis zur Anzahl der CPU-Kerne

```
$ ps -ef
```

listet alle Prozesse am System auf

```
$ ps -ef | grep bzip2
```

listet alle Prozesse mit dem Namen bzip2 auf

```
$ kill -9 PID  (Prozess Identification)
```

beendet Prozess mit der Nummer PID ohne Rücksicht

Etwas im Hintergrund arbeiten lassen: Am Ende des Befehls ein `&` Zeichen: `bzip2 wiki.dump &`

6.5.6 Wegwerfen der Ausgabe einer Datei

`$ /dev/null`

Metazeichen der SHELL um Dinge zu demaskieren:

- `'` umklammern mit Hochkomma
- `\` Backslash vor einzelnen Sonderzeichen

Beispiele:

- `\<ret>` Verlängerungszeichen einer Zeile
- `\<space>` Leerzeichen in Dateinamen

6.5.7 Archive anlegen

Behält die Direktory Struktur, Zeit, Owner und die Zugriffsrechte

```
tar cf archiv.tar file1 file2 file2 ...
```

oder

```
tar cf archiv.tar dir1 dir2 ?
```

6.5.8 Komprimieren von Dateien

```
zip - unzip      .zip
gzip - gunzip    .gz
gunzip -c (zcat)
bzip2 - bunzip2  .bz2
bzcat
7z  7z a 7z x    .7z
```

6.5.9 Archivieren mit Komprimieren

```
tar zcvf archiv.tgz file1 file2 file2 ...
oder
tar zcvf archiv.tgz dir1 dir2 ...
```

Entpacken:

```
tar zxvf archiv.tgz
```

Auflisten des Inhalts eines Archivs

```
tar tf archiv.tar
```

6.5.10 Finden von Dateien und ausführen eines Befehls

`find` zum Extrahieren aller Files eines Verzeichnisbaums und Anwenden eines Befehls

```
$ find . -name "*.txt" -print
```

```
$ find . -name "*.txt" -exec bzcat '{}' \;
| lynx -stdin -dump -display_charset=utf8
```

```
$ find . -name "*.txt" | xargs bzcat
```

6.5.11 Finden von Unterschieden in Dateien

```
$ cmp
```

Vergleicht zwei Dateien Byte für Byte

```
$ diff
```

Vergleicht zwei Dateien Zeile für Zeile

```
$ diff3
```

Vergleicht drei Dateien Zeile für Zeile

6.5.12 UNIX: Tokenisierung, Sortieren, Frequenzlisten

- `tr`
- `grep -o`
- `egrep -o`

6.5.13 Translate von Zeichen

Achtung: bisher (Stand 2016) bietet UNIX beim Befehl `tr` keine utf8 Unterstützung.

```
$ tr -d a < sz.txt
```

löscht alle “a” aus dem Text

```
$ tr -d a < sz.txt
```

löscht alle “a” aus dem Text

```
$ tr " " "\n"
```

löscht alle Spaces und ersetzt sie mit einem Newline(Zeilenumbruch)

```
$ tr -C -d [a-m] < sz.txt
```

```
$ tr [a-z] [A-Z] < sz.txt
```

```
$ tr -s " " "\n"  < sz.txt
```

ersetzt alle wiederholten spaces aus dem Text

6.5.14 Sortieren-einer-wortliste

UNIX Befehl `sort`

- lexikalisch (`sort` ohne parameter)
- nummerisch (`sort -n`)
- aufsteigend (ohne Parameter).
- absteigend (`sort -r`)
- nach Feldern

```
$ tr -s " " "\n" < sz.txt | sort -f # ignoriert Groß/Kleinschreibung
```

```
$ sort -n # numeric sort
```

```
$ sort -f -S 20M sz_utf8.txt
```

```
# mit großem Buffer
### SIZE may be followed by the following multiplicative suffixes: % 1%
### of  memory, b 1, K 1024 (default), and so on for M, G, T, P, E, Z, Y
```

6.5.15 Report und Unterdrücken von Zeilen

```
$ uniq
```

report or omit repeated lines

```
$ uniq -c
```

zählt die Wiederholungszeilen und schreibt die Anzahl davor

```
$ sort -n
```

numerisches Sortieren

```
$ sort -rn
```

numerisches Sortieren, umgekehrte Reihenfolge

```
$ sort -k 2
$ sort --key=2
```

sortieren nach Keys (Feld, welches als Sortierschlüssel dient; wenn nicht angegeben wird gesamte Zeile verwendet)

6.5.16 Erzeugen einer Wortliste

```
$ tr -s " " "\n" < sz_txt.txt | sort | uniq -c | sort -nr | less
oder:
$ cat sz_txt.txt|tr " " "\n"|sort -f|uniq -ic|sort -n> ausgabedatei
```

Die Befehle einzeln:

```
$ cat sz_txt.txt
```

Schreibt die Datei nach STDOUT (Standardausgabe), dort wird sie von der Pipe weitergeleitet.

```
$ tr " " "\n"
```

Übersetzt Leerzeichen in newlines.

```
$ sort -f
```

Ignoriert Groß- und Kleinschreibung für die Sortierung.

```
$ uniq -ic
```

`-c` zählt die einzelnen Tokenvorkommen. `-i` zählt groß- und kleingeschriebene Vorkommen zusammen.

```
$ sort -n
```

Sortiert nummerisch (nach dem Ergebnis von uniq)

```
          > ausgabedatei
```

Leitet in Ausgabedatei um.

```
    174
    153 die
    125 der
    118 und
    100 Linux
     84 von
     57 zu
```

```
56 in
54 für
49 mit
49 den
48 auch
46 eine
(...)
```

Bei diesem Ergebnis handelt es sich um eine einfache Frequenzliste, die noch nicht um newlines, Kommata, Anführungszeichen, etc. bereinigt wurde. Dazu empfiehlt es sich, ein Tool zur Behandlung regulärer Ausdrücke zu verwenden (s.u.).

6.6 Automatisieren von Befehlsfolgen mit der Shell

6.6.1 Was ist ein Shellscript

Eine Datei, die Shell Kommandos enthält, nennt man Shell Script.

In einem Shellscript kann man Linux Befehle, eigene Programme oder andere Shellscripts aufrufen.

Über Ablaufsteuerungsanweisungen, Abfragen und Ausgaben an den Benutzer und den Einsatz von Variablen kann man die Reihenfolge und den Inhalt der aufgerufenen Befehle steuern.

Ein Shellscript wird für eine bestimmte Shell konzipiert (Bourne Shellscript, C-Shellscript, Korn-Shellscript), da sich danach Befehle und Mechanismen richten, die verwendet werden können.

Alle Linux System Shellscripts sind Bourne Shellscripts.

Ein Shellscript wird mit dem Namen gestartet und von einer bestimmten Shell sequentiell interpretiert.

6.6.2 Benutzung von Shellscripts

- komplizierte, sich öfter wiederholende Befehlsfolgen können vereinfacht und so automatisiert werden (Programmerstellung)
- für Systemaufgaben, wie login-Vorgang, Backup, Hochfahren des Systems.
- Die zahlreichen Testmöglichkeiten unter Linux einzusetzen damit bei Kommandofolgen keine Fehler passieren.
- Für Anfänger erleichtern Scripts komplizierte Befehle.

6.6.3 Das Wichtigste in Kürze

- Shellscripts sind in einer Datei zusammengefasste Befehle für eine spezifische Shell
- Es gibt verschiedene Shellscripts
- In Shellscripts kann man Kommentare eintragen, Variablen einsetzen, vom Benutzer Eingaben fordern, Text ausgeben, und mit Kontrollstrukturen arbeiten.
- Die System Shellscripts sind für die Bourne Shell geschrieben.
- Für die Bourne Shell gibt es folgende Kontrollstrukturen:
- if then ϕ
- if then else ϕ
- if then elif ϕ
- for in do done
- while do done
- until do done
- case in esac

6.7 Drucken

6.7.1 CUPS

CUPS, auch für Common Linux Printing System - heute oft das Standarddrucksystem. Die folgenden Befehle sind eher historisch, aus Kompatibilitätsgründen aber in jedem System vorhanden.

Druckaufträge erzeugen (klassisch)

```
lpr text.txt
```

Das Anzeigen von Druckjobs

Zum Ausgeben von Information über den aktuellen Zustand der Drucker, die an das System angeschlossen sind und über den Zustand des Druckauftrags gibt es den Befehl:

```
lpstat
```

Falls ein eigener Druckjob gestartet ist.

bzw.

```
lpstat -t
```

was die gesamte Information über alle Queues und Drucker am System ausgibt. `lpq` ist eine weitere Möglichkeit der Abrage.

Zur Verwaltung von Druckern gibt es ein spezielles Drucker-Kontroll-Programm `lpc`. Es erlaubt den Druckerstatus auszugeben und zu ändern.

Das Löschen von Druckjobs

Jeder Benutzer kann seinen eigenen, der Superuser beliebige Druckaufträge über die Auftragsnummer stoppen :

```
lprm auftragsnummer
```

Das Wichtigste in Kürze

- Drucker werden über spezielle Programme, sogenannte Druckerdämons verwaltet.
- Die Aufgabe des Druckerdämons ist, nur einen Druckauftrag an einem Drucker zu einem Zeitpunkt zuzulassen.
- Der Systemverwalter definiert für jeden Drucker eine Warteschlange.
- Der Benutzer wählt beim Druckbefehl die Warteschlange und somit den Drucker.
- Nachdem der Benutzer den Druckbefehl abgeschickt hat, wird die zu druckende Datei mit den Druckinformationen in das systemweite spool-Verzeichnis kopiert und wenn der Drucker frei ist, vom Druckerdämon zum Drucker geschickt.
- Information über den Drucker und die Druckaufträge bekommt man mit dem Befehl `lpstat -t`. *Jeder Benutzer kann seinen Auftrag über die Auftragsnummer mit dem Befehl* `lprm` *löschen.* Des Printer-Kontroll-Programm `lpc` erlaubt Drucker anzuhalten und neuzustarten.

6.8 Linux-Befehle – eine kurze Übersicht

6.8.1 Dokumentation, Dateien, Pfade

Tabelle 6.3: Linux-Befehle: Dokumentation, Dateien, Pfade

Befehl	wichtige Optionen	Beispiel	Beschreibung
`man`		`man ls`	Hilfe zu einem Befehl (manual)
`info`		`info ls`	Hilfe zu einem Befehl (info)
`whatis`		`whatis ls`	Kurzbeschreibung aus Handbuch
`apropos`		`apropos pwd`	Suche in Handbuch und Kurzbeschreibung
`cd`		`cd /tmp`	wechselt in das genannte Verzeichnis (change directory)
`pwd`		`pwd`	`pwd` gibt das aktuelle Verzeichnis aus (print working directory)
`ls`	`-l -a -R`	`ls -l /tmp`	gibt den Inhalt eines Verzeichnisses (bei `-R` auch Unterverzeichnisse) aus (list); `-l`: langes Format, `-a`: auch versteckte Dateien
`cp`	`-a -r -i`	`cp QUELLE ZIEL`	kopiert Dateien (copy); `-r` mit Unterverzeichnissen; `-a` Archiv-Modus (u.a. Unterverzeichnisse samt Inhalt mit kopieren); `-i` interaktiver Modus, fragt nach vor eventuellem Überschreiben
`mv`	`-i`	`mv QUELLE ZIEL`	verschiebt Dateien oder benennt um (move); `-i` (interaktiv): vor dem Überschreiben von Dateien fragen

Befehl	wichtige Optionen	Beispiel	Beschreibung
`touch`		`touch DATEI`	legt leere Datei an (ändert Zeitstempel vorhandener Dateien)
`rm`	`-f -r`	`rm DATEI`	löscht Dateien (remove); `-f` (force) Löschen erzwingen; `-r` (rekursiv): Verzeichnis samt Inhalt löschen (gefährlich!)
`mkdir`	`-p`	`mkdir -p VERZ/UNTVERZ`	Verzeichnis anlegen (make directory); `-p` (parents): Verzeichnisbaum anlegen
`rmdir`	`-p`	`rmdir VERZ`	Verzeichnis löschen (remove direcotry); `-p` (parents): Verzeichnisbaum löschen, wenn leer
`ln`	`-s`	`ln -s ORIGINAL LINKNAME`	harten Link anlegen; `-s`: symbolischen Link anlegen
`chmod`	`[ugoa][+-=][rwx]`	`chmod go-rwx DATEI`	Lese-(`r`), Schreib-(`w`) und Ausführrecht (`x`) für Besitzer (`u`), Gruppe (`g`), Rest der Welt (`o`) und/oder alle (`a`) hinzufügen (`+`), wegnehmen (`-`) oder exakt setzen (`=`)
`chgrp`		`chgrp users DATEI`	Ändert die Gruppenzugehörigkeit einer Datei; im Beispiel wird `DATEI` der Gruppe users zugeordnet.
`chown`		`chown root DATEI`	Ändert die Eigentümer einer Datei; im Beispiel wird `DATEI` dem Benutzer root zugeordnet
`getfacl`		`getfacl DATEI`	zeigt ACL-Zugriffsrechte von Datei an

Befehl	wichtige Optionen	Beispiel	Beschreibung
`setfacl`	`-m -x`	`setfacl -m u:USER:r DATEI`	setzt ACL-Zugriffsrechte von Datei; im Beispiel werden der Person `USER` Leserechte (`r`) auf `DATEI` gegeben; `-m`: Rechte setzen; `-x`: Rechte löschen
`getfattr`		`getfattr -d -m '-' DATEI`	zeigt erweiterte Attribute (capabilities)von Datei an
`setfattr`			setzt erweiterte Attribute von Datei
`setcap`			setzt capabilities
`getcap`		`/sbin/getcap /usr/bin/ping`	zeigt cabailities an, Beispiel nicht auf allen System genutzt, ubuntu nutzt s-flag stattdessen

6.9 Textverarbeitung

Tabelle 6.4: Linux-Befehle: Textverarbeitung

Befehl	wichtige Optionen	Beispiel	Beschreibung
`cat`	`-n`	`cat DATEI`	Dateiinhalt ausgeben (concatenate); `-n` Zeilen numerieren
`head`, `tail`	`-n`	`head -n 100 DATEI`	gibt die ersten/letzten 100 Zeilen von DATEI aus. `tail -f` fortlaufend
`wc`	`-c -m -l -w`	`wc -l DATEI`	gibt die Anzahl der Bytes (`-c`), Zeichen (`-m`), Zeilen (`-l`) oder Wörter (`-w`) von `DATEI` aus
`more`, `less`	`-S`	`less -S DATEI`	zeigt eine Datei seitenweise an; `-S`: Zeilen nicht umbrechen (nur less)
`sort`	`-r -n`	`sort DATEI`	sortiert eine Datei; `-r`: rückwärts sortieren; `-n` numerisch sortieren
`grep`	`-i -l -v -r`	`grep -i regexp DATEI(EN)`	durchsucht Datei(en) nach einem regulären Ausdruck und gibt passende Zeilen aus; `-i` (ignore case) Groß-/Kleinschreibung ignorieren, `-l` (list): lediglich Dateien mit Treffern anzeigen; `-v`: (invert match): nur nicht-passende Zeilen ausgeben; `-r`: alle Dateien unter einem Verzeichnis rekursiv durchsuchen
`agrep`	`-#`	`agrep -2 "hallo" DATEI`	Findet “hallo” oder ähnliche Schreibweisen in Datei; über `-#` wird der Levenshtein-Abstand eingestellt
`tr`		`tr "a" "b"`	Zeichen ersetzen. Im Beispiel wird jedes “`a`”, das von stdin gelesen wird, durch “`b`” ersetzt.
`sed`	`s`	`sed s/regexp/replace/`	ersetzt regulären Ausdruck durch Replacement
`uniq`	`-c`	`uniq -c INDATEI AUSDATEI`	Löscht alle doppelten Zeilen aus INDATEI und schreibt das Ergebnis nach AUSDATEI ; `-c`: Anzahl der Vorkommen angeben
`diff`	`-r -y -u`	`diff ALT NEU`	vergleicht 2 Dateien (difference); `-r`: Verzeichnisse rekursiv vergleichen
`cut`	`-d -f`	`cut -d "," -f1 DATEI`	extrahiert spaltenweise Ausschnitte aus Textzeilen; `-d`: Angabe der Trennzeichen; `-f`: Angabe der zu extrahierenden Felder

Befehl	wichtige Optionen	Beispiel	Beschreibung
`join`	`-e`	`join DATEI1 DATEI2`	verknüpft Zeilen mit identischen Feldern; `-e`: ersetzt fehlende Felder durch festgelegte Zeichenkette

6.10 Komprimieren von Dateien/Ordnern

Tabelle 6.5: Linux-Befehle: Komprimieren von Dateien/Ordnern

Befehl	wichtige Optionen	Beispiel	Beschreibung
`tar`	`[t|x|c][j|z]f`	`tar xvjf DATEI.tar.bz2`	.tar.bz2-Archiv anzeigen (test), auspacken (extract) oder anlegen (create) (tape archiver). `z` bei gzip-komprimierten Dateien (.tgz oder .tar.gz), `j` bei bzip2-komprimierten Dateien (.tbz oder .tar.bz2)
`gunzip`, `gzip`, `zcat`	`-l`	`gunzip -l DATEI.gz`	.gz-Datei auspacken oder Inhalt auflisten (`-l`); gzip: Datei packen; zcat Datei auspacken und auf stdout schreiben
`unzip`, `zip`	`-l`	`unzip DATEI.zip`	.zip-Datei auspacken oder Inhalt auflisten (`-l`); zip: zip-Datei packen
`bzip2`		`bzip2 DATEI`	Packt `DATEI` nach `DATEI.bz2`
`7z`	`x`	`7z x DATEI.7z`	Entpackt Dateien aus dem Archiv `DATEI.7z`
`split`	`-b`	`split -b=650m DATEI`	Teilt `DATEI` in mehrere 650MB große Stücke, zusammensetzen mit `cat`

6.11 Dateisystem

Tabelle 6.6: Linux-Befehle: Dateisystem

Befehl	wichtige Optionen	Beispiel	Beschreibung
`locate`		`locate *.pl`	Dateien lokalisieren (benötigt regelmäßige Läufe von updatedb), hier werden alle Dateien mit Endung `.pl` angezeigt
`find`		`find -name "foo*"`	aktuelles Verzeichnis und Unterverzeichnisse nach Dateien durchsuchen, deren Name auf ein Muster passt
`mount`, `unmount`		`mount /dev/sdb1/ /mnt/sdb1`	Laufwerk einbinden/aushängen (mit Root-Rechten)
`gio`	`mount -d -e`	`gio mount -d /dev/sdb1 gio mount -e /run/media/user/usb-stick`	bindet Laufwerke mittels Gnome Virtual File System (GVFS) ein und aus(ohne Root-Rechte)
`du`	`-s -h`	`du -s`	zeigt den durch die Dateien eines Verzeichnisses und seine Unterverzeichnisse belegten Platz an (disk usage); `-s` (summary): lediglich Gesamtwert, nicht jedes einzelne Unterverzeichnis ausgeben; `-h`: sinnvolle Größenangabe ; alterntiv: `ncdu`
`df`	`-h`	`df -h .`	Zeigt die Belegung der gemounteten Laufwerke an; `-h`: sinnvolle Größenangabe

6.12 Prozessmanagement und User

Tabelle 6.7: Linux-Befehle: Prozessmanagement und User

Befehl	wichtige Optionen	Beispiel	Beschreibung
`su`	`-`	`su - kennung`	Shell mit anderer Benutzerkennung starten (root, wenn kein Benutzer genannt ist); `-`: Login-Shell starten
`sudo`		`sudo /sbin/halt`	Program mit root-Rechten ausführen (super user do)
`ps`	`aux`	`ps aux`	Prozessliste anzeigen (processes), zeigt PIDs zu Prozessen an
`kill`	`-9 -l`	`kill PID`	Signal an den Prozess mit der Prozess-ID PID schicken, Standard: SIGTERM (termination); `-9`: SIGKILL- Signal schicken
`killall`	`-9`	`killall find`	Prozess über seinen Namen beenden
`nice`	`-n`	`nice -n 10 BEFEHL`	Startet Befehl mit Nettigkeit `10`. Je größer die Zahl desto größer die Nettigkeit, `0` normale, `-20` niedrigste, `19` höchste Nettigkeit)
`renice`		`renice # PID`	Setzt für PID die Nettigkeit `#`, `#` im Intervall [`-20,19`]
`top`		`top`	zeigt eine selbstaktualisierende Tabelle mit Prozessinformationen an (q für quit)
`bg`, `fg`		`bg %1`	setzt den Prozess %1 fort (%1 ist die Nummer des Prozesses, der mit Strg+z unterbrochen wurde)
`last`	`-a -i`	`last -a -i`	Benutzer anzeigen, die zuletzt auf diesem Rechner waren.
`who`, `w`			anzeigen wer eingeloggt ist (`who`). `w` zeigt zusätzlich Informationen über laufende Prozesse an
`finger`		`finger USER`	`finger` zeigt Kontaktinformationen über einen Benutzer an

6.13 Zeichensätze

Tabelle 6.8: Linux-Befehle: Zeichensätze

Befehl	wichtige Optionen	Beispiel	Beschreibung
`recode`	`-l`	`recode latin-1..utf-8 DATEI.txt`	Konvertiert eine Datei zwischen verschiedenen Zeichensatzformaten
`iconv`	`-l -f -t`	`iconv -f l1 -t utf8 DATEI.txt`	Konvertiert eine Datei zwischen verschiedenen Zeichensatzformaten
`uniconv`	`-decode -encode -l`	`uniconv -decode utf-8 -encode German`	Konvertiert zwischen verschiedenen Zeichsatzformaten
`locale`	`-a`	`locale`	zeigt landesspezifische Umgebungsvariablen an
`file`	`-i`	`file DATEI`	Dateityp erraten; `-i`: Mime Typ ausgeben
`hexdump`		`hexdump -C UTF-8-demo.utf | less`	zeigt Daten byteweise hexadezimaler Form an
`od`	`-b`		zeigt Daten in oktaler Form an

6.14 Umgebungsvariablen

Tabelle 6.9: Linux-Befehle: Umgebungsvariablen

Befehl	wichtige Optionen	Beispiel	Beschreibung
`export`		`export LANG="de_DE@euro"`	setzt Variablen, zeigt ohne Argumente alle Umgebungsvariablen an
`set`			zeigt Umgebungsvariablen an
`unset`			löscht Umgebungsvariable
`setenv`			wie export, für andere shells
`echo`	`-n`	`echo $PATH`	Variable ausgeben; `-n` (newline): kein Zeilenvorschub nach Ausgabe

6.15 Netzwerkeigenschaften

Tabelle 6.10: Linux-Befehle: Netzwerkeigenschaften

Befehl	wichtige Optionen	Beispiel	Beschreibung
`ss`	`-p`	`ss -p`	sockets statistics zeigt Informationen über das Netzwerk an; `-p`: Anzeige der dazugehörigen Prozesse
`netstat`	`-a -n -p -T -W`	`netstat -anpT`	Netzwerkstatus, ähnlich socket statistics
`ping`	`-6 -4 -c N`	`ping www.lrz.de`	Host anpingen (ohne Option `-c` mit Strg+c abbrechen)
`ip`	`-6 -4 a r m n l`	`ip a`	Anzeige der IP Adressen, Routen, Multicastadressen, Nachbarn u.v.a.m.

6.16 Via Internet Dateien kopieren/Rechner fernsteuern

Tabelle 6.11: Linux-Befehle: Via Internet Dateien kopieren/Rechner fernsteuern

Befehl	wichtige Optionen	Beispiel	Beschreibung
`wget`	`-p -r -l`	`wget -p http://www.lmu.de`	Internetseite auf Festplatte kopieren; `-p`: alle zur Darstellung der Seite relevanten Dateien mitladen; `-r`: Seite rekursiv `-l`: Rekursionstiefe begrenzen
`curl`	`-O`	`curl http://www.lmu.de`	Internetseite auf Terminal anzeigen; `-O`: schreibt Output in Datei
`ssh`	`-X`	`ssh -X login@ssh.cip. ifi.lmu.de`	ssh-Verbindung zu Server aufbauen; Fernsteuerung per Kommandozeile
`scp`		`scp login@ssh.cip. ifi.lmu.de:DATEI ZIEL`	Kopiert die Datei von der Uni in lokalen Ordner

Befehl	wichtige Optionen	Beispiel	Beschreibung
		`scp DATEI login@ssh.cip ifi.lmu.de:ZIEL`	Kopiert die Datei vom lokalen Ordner in einen Remote-Ordner
`rsync`	`-a --delete -n`	`rsync -av LocalDir/ user@host.domain.tld:/RemoteDir/`	Synchronisiert Verzeichnisse via Internet
`sftp`		`sftp login@sftp.home.lan`	sftp-Verbindung zu Server aufbauen
`rdesktop`		`rdesktop -g 80% remote.cip.ifi.lmu.de`	baut Remote-Desktop-Verbindung auf; Fernsteuerung mit grafischer Oberfläche
`wlfreerdp`	`/v:<server>[:port]`	`wlfreerdp /v:remote.cip.ifi.lmu.de`	baut Remote-Desktop-Verbindung auf; Fernsteuerung mit grafischer Oberfläche (wayland client)
`vncviewer`		`vncviewer vnc.cis.lmu.de:1`	Zugriff auf entfernten Rechner / Virtual Network Computing, Fernsteuerung mit GUI

6.17 Operatoren zur Ausgabe-/Eingabeumleitung

Tabelle 6.12: Linux-Befehle: Operatoren zur Ausgabe-/Eingabeumleitung

Operator/Befehl	wichtige Optionen	Beispiel	Beschreibung
`<`		`tr -s "a" < datei`	stdin umleiten
`>`		`ls > datei.txt`	stdout umleiten
`2>`		`befehl 2>&1`	stderr umleiten
`>>`		`ls >> DATEI.txt`	stdout an Datei anhängen
|		`cat sz.txt |less`	an Prozess weiterleiten
`` ` ` ``			mit Ergebnis des Programmaufrufs weiterarbeiten (‘Befehl‘)
`tee`			liest von stdin und schreibt zu stdout und Dateien

6.18 Versionsverwaltung für Quellcode und Dokumentation

Tabelle 6.13: Linux-Befehle: Versionsverwaltung für Quellcode und Dokumentation

Befehl	wichtige Unterbefehle	Beispiel	Beschreibung
`svn`	`co ci add up`	svn co	Dateien mittels svn runterladen (`co`,`up`), hochladen (`ci`) oder zum Repository hinzufügen (`add`).
`git`	`clone add push pull`	git pull	Dateien mittels git runterladen (`pull`), hochladen (`push`) oder zum Repository hinzufügen (`add`).

6.19 Sonstiges

Tabelle 6.14: Linux-Befehle: Sonstiges

Befehl	wichtige Optionen	Beispiel	Beschreibung
`screen`	`-r`	`screen befehl`; `screen`	parkt aktive Sessions, wichtige Tastenkombination: ctrl+a, d, c, a
`lpr`, `lprm`, `lpq`		`lpr DATEI`	`lpr` druckt datei auf einem Drucker; `lprm` löscht Druckjobs, `lpq` zeigt Druckjobs an
`crontab`	`-e -l -r`	`crontab -e`	Crontab bearbeiten (`-e`), anzeigen (`-l`), oder löschen (`-r`)
`at`, `atrm`, `atq`		`echo "reboot" | at 5:00`	Befehl zu bestimmtem Zeitpunkt ausführen
`clear`, `reset`		`reset`	Terminal wiederherstellen
`wodim`		`wodim -v dev=/dev/sr0 image.iso`	brennt iso.images
`mkisofs`, `genisoimage`	`-J -r -o`	`genisoimage -o image.iso pfad`	erzeugt CD/DVD image-Datei
`which`		`which rm`	kompletten Pfad eines Programmes in $PATH ausgeben

7 Unicode Properties in Regulären Ausdrücken

Unicode führt sogenannte Properties ein, die mit einem `\p` un den in geschweiften Klammern eingeschlossenen Eigenschaften notiert werden. Verwendet man als Unicode Property nicht den Kleinbuchstaben `p` sondern den Großbuchstaben `P`, dann spricht bedeutet dies die Negation der in geschweiften Klammern angegebenen Eigenschaft. Eine weitere Möglichkeite eine Eigenschaft zu negieren, ist das Negationszeichen `^` der Eigenschaft vorzustellen.

Das `re` Modul mit dem in Kapitel 3 gearbeitet wurde unterstützt diese Unicode Properties nicht. Um diese verwenden zu können, muss das `regex` Modul installiert sein. (https://pypi.python.org/pypi/regex)

		Beschreibung
`\p{L}`	`\p{Letter}`	Jeder Buchstabe aus jeder Sprache.
`\p{Ll}`	`\p{Lowercase_Letter}`	Ein kleingeschriebener Buchstabe, welcher auch eine großgeschriebene Variante hat.
`\p{Lu}`	`\p{Uppercase_Letter}`	Ein großgeschriebener Buchstabe, welcher auch eine kleingeschriebene Variante hat.
`\p{Lt}`	`\p{Titlecase_Letter}`	Ein großgeschriebener Buchstabe am Anfang eines Wortes, wenn das Wort sonst nur aus kleingeschriebenen Buchstaben besteht.
`\p{L&}`	`\p{Cased_Letter}`	Ein Buchstabe, der groß- und kleingeschriebene Varianten hat (Kombination von Ll, Lu und Lt).
`\p{Lm}`	`\p{Modifier_Letter}`	Ein spezielles Zeichen, welches wie ein Buchstabe verwendet wird.
`\p{Lo}`	`\p{Other_Letter}`	Ein Buchstabe oder ein Ideogram welche keine kleingeschriebene Version haben.
`\p{M}`	`\p{Mark}`	Ein Zeichen, welches dazu gedacht ist, mit einem anderen Zeichen kombiniert zu werden.

		Beschreibung
`\p{Mn}`	`\p{Non_Spacing_Mark}`	Ein Zeichen, welches dazu gedacht ist, mit einem anderen Zeichen kombiniert zu werden, ohne mehr Platz einzunehmen.
`\p{Mc}`	`\p{Spacing_Combining_Mark}`	Ein Zeichen, welches dazu gedacht ist, mit einem anderen Zeichen kombiniert zu werden und mehr Platz einnimmt.
`\p{Me}`	`\p{Enclosing_Mark}`	Ein Zeichen, welches das Zeichen, mit dem es kombiniert wird umschließt.
`\p{Z}`	`\p{Separator}`	Jede Art von Leerzeichen oder unischtbaren Zeichen.
`\p{Zs}`	`\p{Space_Separator}`	Ein Leerzeichen, das unsichtbar ist, aber Platz einnimmt.
`\p{S}`	`\p{Symbol}`	Mathematische Symbole, Währungszeichen, Zierrate (dingbats), etc.
`\p{Sm}`	`\p{Math_Symbol}`	Ein beliebiges mathematisches Symbol.
`\p{Sc}`	`\p{Currency_Symbol}`	Ein beliebieges Währungszeichen.
`\p{Sk}`	`\p{Modifier_Symbol}`	Ein Zeichen, welches dazu gedacht ist, mit einem anderen Zeichen kombiniert zu werden, aber alleine steht.
`\p{So}`	`\p{Other_Symbol}`	Diverse Symbolde, welche nicht matschematische Symbole, Währungszeichen oder Kombinierzeichen sind.
`\p{N}`	`\p{Number}`	Ein beliebiges numerisches Zeichen in jedem beliebigen Skript.
`\p{Nd}`	`\p{Decimal_Digit_Number}`	Eine beliebige Ziffer von 0 bis 9 in jedem beliebigen Skript, ausgenommen ideographische Skripte.
`\p{Nl}`	`\p{Letter_Number}`	Eine Nummer, die wie ein Buchstabe aussieht (z.B.: Römische Zahlen)
`\p{No}`	`\p{Other_Number}`	Eine hoch- oder tiefgestellte Ziffer oder eine Nummer, welche keine Ziffer von 0 bis 9 ist (ausgenommen Nummern von ideografischen Skripten).
`\p{P}`	`\p{Punctuation}`	Ein beliebieges Punktuationszeichen.
`\p{Pd}`	`\p{Dash_Punctuation}`	Ein beliebiger Binde- oder Gedankenstrich.
`\p{Ps}`	`\p{Open_Punctuation}`	Eine beliebige öffnende Klammer.
`\p{Pe}`	`\p{Close_Punctuation}`	Eine beliebige schließende Klammer.
`\p{Pi}`	`\p{Initial_Punctuation}`	Ein beliebiges öffnendes Anführungszeichen.

		Beschreibung
`\p{Pf}`	`\p{Final_Punctuation}`	Ein beliebiges schließendes Anführungszeichen.
`\p{Pc}`	`\p{Connector_Punctuation}`	Ein Punktuationszeichen, wie der Unterstrich, welches Wörter verbindet.
`\p{Po}`	`\p{Other_Punctuation}`	Ein beliebiges Punktuationszeichen, welches nicht Bindestrich oder Gedankenstrich, Klammer, Anführungszeichen, oder Verbinder ist.

Die folgenden Beispiele gehen davon aus, dass das `regex` Modul installiert (`pip3.5 install regex`) und importiert (`import regex`) worden ist.

```
>>> text = "123abc456"
>>> p = regex.compile('\p{L}')
>>> regex.findall(p, text)
['a', 'b', 'c']
>>>
>>> text = "4€ Bagel, 1.90€ Espresso, 4.10€ Smoothie"
>>> p = regex.compile('\p{N}+\.?\p{N}{0,2}\p{Sc}')
>>> regex.findall(p, text)
['4€', '1.90€', '4.10€']
```

8 Ausgewählte Beispiele

8.1 Erzeugen einer Frequenzliste von Wörtern aus einer Datei

Schreiben Sie ein Programm, das den Text aus der Datei `eingabe.txt` liest und erzeugen Sie eine Frequenzliste (dictionary) aus den Wörtern. Speichern Sie die Einträge der Wortliste in die Datei 'frequenzliste.txt'.
Das Format der einzelnen Zeile in der Wortliste frequenzliste.txt ist:

Das Wort ... kommt ... mal vor.

```
#!/usr/bin/python3
#Aufgabe 8-3
#Autorin: Leonie Weißweiler, WD 2016/17

text = open('Verbrechen_Strafe_1924.txt', 'r') #öffnet die Datei zum Lesen

#hier wird ein leeres dictionary erzeugt,
#damit für Python der Datentyp der Variable frequenzliste klar ist
frequenzliste = {}

#allewoerter wird als leere Liste initialisiert,
#damit wir gleich an sie anhängen können
allewoerter=[]

#######################
#iteriert über die Zeilen der Datei
for line in text:

    for word in line.split(' '): #iteriert über die Wörter der Zeile

        word = word.strip() #entfernt whitespace am Anfan/Ende des Worts

        word = word.lower() #konvertiert in Kleinbuchstaben

        #überprüft, ob der key word im dictionary frequenzliste vorkommt
        if (word in frequenzliste):
            #falls ja, wird der value des keys word um eins erhöht,
```

```
            #weil wir ja wieder ein Vorkommen gefunden haben
            frequenzliste[word] = frequenzliste[word] + 1


            #falls der key word noch nicht im dictionary enthalten war
            #wird er neu erzeugt, indem der Value auf 1 gesetzt wird
            #(es war ja das erste Vorkommen des Wortes)
        else:
            frequenzliste[word] = 1

########################################
#sortieren der Wörter nach der Frequenz
#iteriert durch die absteigend nach value sortierten Paare
for wort, frequenz in sorted(frequenzliste.items(),
                             key=lambda x: x[1],reverse=True):

    #hängt den jeweiligen key, also das Wort, an die Liste allewoerter an
    allewoerter.append(wort)


#gibt jeweils das Wort auf dem Index x aus, und zwar die ersten zehn,
#also die zehn häufigsten
#iteriert von 0 bis ausschließlich 10
for x in range(0,10):

#gibt jeweils das Wort auf dem Index x aus, und zwar die ersten zehn,
#also die zehn häufigsten
#iteriert von 0 bis ausschließlich 10 und gibt die Wörter und
#ihre Frequenz aus.
for x in range(0,10):

    print ("Das Wort",allewoerter[x]," kommt ",
                      frequenzliste[allewoerter[x]], " mal vor")

#schließt das filehandle
text.close()
```

9 Bibliographie

H.-P. Gumm, M. Sommer. {1994}. *Einführung in Die Informatik*. Addison-Wesley, Bonn. 579 S., 359 Abb., Hardcover.

Herbert Klaeren, Michael Sperber. {2001}. *Vom Problem Zum Programm. Architektur Und Bedeutung von Computerprogrammen*.