

Mathe-Notizen für Selbststudium

1. Auflage

vom 9. Juli 2017



Jens Kallup

Langensalzer Str. 30
99817 Eisenach
Tel.: 03691 /
E-Mail: jkallup@web.de

Inhalt:

Eigene Gedanken zu:
de.sci.mathematik
de.sci.informatik

Inhaltsverzeichnis

1	Vorwort	11
2	Grundlagen - Informatik	13
2.1	Grundlagen	13
2.2	Programmieren mit Python	13
2.2.1	Der Interpreter - Interaktivmodus	14
2.2.2	Zeichenketten	15
2.2.3	Ausführen von Python-Code	15
2.2.4	Python Interna	16
2.2.5	Compiler	17
2.2.6	Interpreter	17
2.2.7	Strukturierung durch Einrückung	17
2.2.8	Datentypen und Variablen	18
2.2.9	Zahlen	18
2.2.10	Zeichenketten - Strings	19
2.2.11	Unveränderliche Zeichenketten	20
2.2.12	Escape- oder Fluchtzeichen	20
2.2.13	Typwechsel bei Variablen	21
2.2.14	Wechselnde Speicherorte	22
2.2.15	Besonderheiten bei Strings	23
2.3	Ausdrücke und Operatoren	23
2.3.1	Operatoren	23
2.4	Tiefes und flaches Kopieren	24
2.4.1	Kopieren einer Liste	24
2.4.2	Kopie mit der Methode deepcopy aus dem Modul copy	26
2.5	Bedingte Anweisungen	26
2.5.1	Die if-Anweisung	27
2.5.2	Beispiel Hundejahre	27
2.5.3	Wahr oder falsch	27
2.5.4	Abgekürztes IF	28
2.5.5	Steuerrechner in Python	28
2.6	Eingabe mit input()	29
2.7	Eingabe mit raw_input()	30
2.8	Schleifen	31
2.8.1	Standard-Eingabe lesen	32
2.8.2	Der else-Teil	33
2.8.3	Vorzeitiger Abbruch einer while-Schleife	33
2.9	for-Schleifen	34
2.10	Die range()-Funktion	35
2.11	Iteration über Liste mit range()	37

2.12	Listen-Iteration mit Seiteneffekten	38
2.13	Ausgabe mit print	38
2.13.1	Import aus der Zukunft: print_function	39
3	Grundlagen - Mathematik	41
3.1	Zahlensysteme	41
3.1.1	Dualsystem	41
3.2	Umwandlung Dezimal - Dual	41
3.2.1	Dezimal nach Dual umwandeln	41
3.2.2	Dual in Dezimalzahl umwandeln	42
3.3	Hexadezimalsystem	42
3.3.1	Umwandlung Hex nach Dezimal	42
3.4	Zeichen und Symbole	43
4	Zahlen und Zahlenbereiche	45
4.1	Natürliche Zahlen	45
4.1.1	Einführung	45
4.2	ganze Zahlen	45
4.2.1	Einführung	45
4.3	rationale Zahlen	46
4.3.1	Einführung	46
4.4	gebrochene Zahlen	46
4.4.1	Einführung	46
4.5	reelle Zahlen	47
4.5.1	Einführung	47
4.5.2	Die Quadratwurzel aus 2 ist keine rationale Zahl	47
4.6	komplexe Zahlen	48
4.6.1	Einführung	48
4.7	Beispiel aus de.sci.mathematik	48
5	Mengen	51
6	Funktionen	53
7	Neuronale Netze	57
7.1	Arten von Netzwerken	57
7.2	Was sind Neuronale Netzwerke ?	57
7.2.1	Künstliches Neuron	57
7.2.2	Biologische Motivation	57
7.2.3	Modellierung	58
7.2.4	Bestandteile	59
7.3	Mathematische Definition	60
7.3.1	on-Neuron	60
7.4	Merkmalsraum	60
7.4.1	Merkmalsvektor	61
7.4.2	Musterklassifikation	61

7.4.3	Klassifikator	61
7.4.4	Klassifikationsverfahren	62
8	Überwachtes Lernen	65
8.1	Empirische Daten	65
8.2	mittlerer quadratische Fehler	66
8.3	Punktschätzer	66
8.4	Schätzfunktion	66
8.5	Grundgesamtheit	66
8.6	Stichprobe	66
8.6.1	Auswahlverfahren	67
8.7	Systematische Stichprobe	67
8.8	Stichprobenvariable	67
8.9	Stichprobenfunktion	68
8.9.1	Arithmetisches Mittel - Formel	68
8.9.2	Stichprobenfunktion - Formel	68
8.10	Stichprobenverteilung	68
8.11	Hypothese	68
8.11.1	Alternativhypothese	68
8.12	Teststatistik	69
8.12.1	Statistische Signifikanz	69
8.12.2	Verwendung bei festem Signifikanzniveau	69
8.12.3	Fehler 1. Art	69
9	Aktivierungsfunktionen	71
9.1	Schwellwertfunktion	71
9.2	Stückweise lineare Funktion	71
9.3	Sigmoid Funktion	71
9.4	Darstellung boolescher Funktionen	72
10	Formelsammlung	73
10.1	Umrechnungen	73
10.2	mathematische Formeln	74
A	Anhang	75

Tabellenverzeichnis

Abbildungsverzeichnis

2.1	Pythagoras-Satz	37
7.1	Nervenzelle	58
7.2	künstliches Neuron	59
7.3	input neuron	60
9.1	Graph der Sigmoid-Funktion	72

Vorwort

mit diesen Posting will ich versuchen, eine Zusammenfassung dessen zu geben, was hier seit Monaten Diskutiert wird. Falls was falsch sein sollte, bitte Feedback geben. Kann auch passieren das ich aus versehen eine falsche Taste beim schreiben drücke, und der Inhalt fehlt, ich bemühe mich durchgängig zu schreiben und im Zusammenhang zu posten. Ok, let's go...

Grundlagen - Informatik

2.1 Grundlagen

Die zum Erledigen einer Aufgabe erforderlichen Arbeitsschritte lassen einen immer wiederkehrenden Rhythmus erkennen:

E - ingabe

V - erarbeitung

A - usgabe .

Der Aufbau der Zentraleinheit aus elektronischen Schaltungen bringt es mit sich, dass zur Datendarstellung nur zwei Zustände gegeben sind.

”Ja” - Strom fließt und

”Nein” - kein Strom.

Diese beiden Zustände werden mit **1** oder **0** bezeichnet.

Das Bit ist ein Binärzeichen, das die Zustände 1 und 0 annehmen kann. Es ist zugleich die kleinste Informationseinheit, die Computer verstehen. Alle Informationen müssen auf dem Bit aufgebaut werden.

Hinweis: Für Grundlagen der in der Informatik verwendeten Zahlensysteme, finden Sie auch im Kapitel 3 - 3.1 auf Seite 41 .

Für die Programmierung von Programmen, die in der Informatik Anwendung finden müssen bestimmte Voraussetzungen als Grundlage gegeben sein.

Hier finden Sie Information zur Sprache Python:

2.2 Programmieren mit Python

Die Sprache wurde Anfang der 1990er Jahre von Guido van Rossum am Zentrum für Mathematik (Centrum voor Wiskunde en Informatica) in Amsterdam entwickelt. Ursprünglich war sie als Nachfolger für die Lehrsprache ABC entwickelt worden und sollte auf dem verteilten Betriebssystem Amoeba laufen. Guido van Rossum hatte auch an der Entwicklung der Sprache ABC mitgewirkt, so dass seine Erfahrungen mit ABC auch in Python einfließen.

Auch wenn wir auf dieser Webseite nicht mit Python-Schlangen geizen, hat der Name der Programmiersprache Python nichts mit den Schlangen zu tun. Für Guido van Rossum stand vielmehr die britische Komikertruppe Monty Python mit ihrem legendären Flying Circus Pate für den Namen.

Guido van Rossum schrieb 1996 über die Entstehung des Names seiner Programmiersprache: „Vor über sechs Jahren, im Dezember 1989, suchte ich nach einem 'Hobby'-Programmier-Projekt, dass mich über die Woche um Weihnachten beschäftigen konnte. Mein Büro ... war zwar geschlossen,

aber ich hatte einen PC und sonst nichts vor. Ich entschloss mich einen Interpreter für die neue Skripting-Sprache zu schreiben, über die ich in der letzten Zeit nachgedacht hatte: ein Abkömmling von ABC, der UNIX/C-Hackern gefallen würde. Python hatte ich als Arbeitstitel für das Projekt gewählt, weil ich in einer leicht respektlosen Stimmung war (und ein großer Fan von Monty Python's Flying Circus)."

Dennoch sind Assoziationen mit Schlangen möglich und sinnvoll: Man denke nur an das Python-Toolkit "Boa" oder die Programmiersprache Cobra.

2.2.1 Der Interpreter - Interaktivmodus

Der Python-Interpreter ist Bestandteil fast jeder Linux-Distribution und befindet sich meist im Pfad `/usr/bin/` (auch `/usr/local/bin/`).

Man kann Python im interaktiven Modus aufrufen, indem man den Interpreter in einer Shell ohne Parameter aufruft.

Python meldet sich mit Informationen über die installierte Version:

```
$ python
Python 2.5.2 (r252:60911, Oct 5 2008, 19:29:17)
[ GCC 4.3.2 ] on linux2
"help", "copyright", "credits" or "license" for more information.
=>
```

Nach dem Eingabeprompt (`=>`) kann beliebiger Python-Code eingegeben werden, der nach dem Quittieren mit der Eingabetaste sofort ausgeführt wird. So kann man den Interpreter beispielsweise als „Taschenrechner“ benutzen:

```
=> 4.567 * 8.323 * 17
646.18939699999999
=>
```

Das folgende Ergebnis mag den einen oder die andere überraschen:

```
=> 12 / 7
1
=>
```

Python nimmt an, dass man an einer Integer-Division interessiert ist, weil sowohl der Divisor als auch der Divident Integer-Werte sind. Deshalb erhalten wir auch einen Integer-Wert als Resultat. Der einfachste Weg ein Ergebnis mit Nachkommazahlen zu erhalten besteht darin, einer der beiden Werte mit `".0"` zu erweitern:

```
=> 12 / 7
1.7142857142857142
=>
```

Alternativ kann man einen oder beide Werte mit einer `cast`-Funktion wandeln:

```
=> float(12) / 7
1.7142857142857142
=>
```

Python befolgt Punktrechnung (Division und Multiplikation) geht vor Strichrechnung (Addition und Subtraktion). Das bedeutet, dass man den folgenden Ausdruck auch ohne Klammern schreiben

kann: `"3 + (2 * 4)":`

```
⇒ 3 + 2 * 4
11
⇒
```

Der jeweils letzte Ausgabewert wird vom Interpreter in einer speziellen Variablen automatisch gespeichert. Der Name der Variable ist einfach ein Unterstrich (underscore), also `"_"`. Das Ergebnis der letzten Berechnung kann man sich also wieder ausgeben lassen:

```
⇒ _
11
⇒
```

Der Unterstrich kann im Prinzip wie eine normale Variable benutzt werden:

```
⇒ _ * 3
33
⇒
```

Den Python-Interpreter kann man mit Ctrl-D unter Linux wieder verlassen.

Möchte man ein Skript aus einer Datei einlesen und ausführen, so kann man dies mit dem `execfile()`-Kommando tun. Um das nachfolgende Beispiel nachvollziehen zu können, kann man die Anweisung

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

in einer Datei unter dem Namen `„hello.py“` abspeichern.

```
⇒ execfile("hello.py")
Hello World
⇒
```

2.2.2 Zeichenketten

Zeichenketten, die auch im Deutschen meist besser als Strings bekannt sind, kann man in Python ganz einfach mit doppelten Hochkommata (Anmerkung: geht auch mit einfachen oder Dreifachen Hochkommas) erzeugen. Wie bei den Zahlen gibt es auch für die Strings Operatoren.

```
⇒ "Hello" + " " + "World"
'Hello World'
⇒
```

Es gibt auch eine "Multiplikation" auf Strings:

```
⇒ ".-" * 4
'-.-.-.-.-'
⇒
```

2.2.3 Ausführen von Python-Code

Betrachten wir folgende Python-Code-Zeile:

```
print "Python lernen!"
```

Man erkennt unschwer die Variante zum sonst nahezu obligatorischen Hallo-Welt bzw. Hello-World-Skript. Diese Anweisung gibt also `"Python lernen"` aus, wie man erkennt, wenn man die Zeile in

der Shell des Python-Interpreter eintippt:

```
⇒ print "Python lernen!"
Python lernen!
```

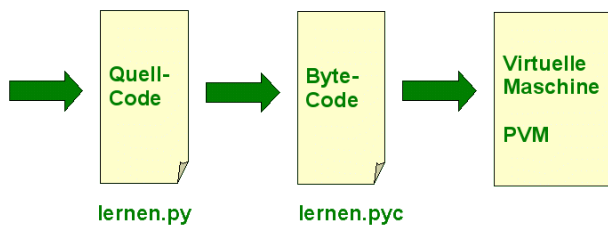
Die meisten Skripte werden jedoch nicht interaktiv eingetippt und ausgeführt sondern, wie auch in anderen Programmiersprachen üblich in einer Datei gespeichert. Obiges Skript könnte man beispielsweise unter dem Namen `python_lernen.py` speichern. Um ein Python Programm zu erstellen, benötigt man keine IDE ("Integrated Development Environment"), wie zum Beispiel IDLE. Im Prinzip kann man jeden Editor (`vi`, `emacs` und andere) verwenden, der in der Lage ist unformatierte Textdateien abzuspeichern.

Das Suffix `.py` ist für ein einzelnes Skript in Python nicht unbedingt nötig. Zur Kennzeichnung von Modulen ist es jedoch zwingend nötig, wie man im Kapitel "Module und Pakete" nachlesen kann.

2.2.4 Python Interna

Nun wollen wir die Frage klären, was innerhalb von Python nach dem Start eines Skriptes geschieht. Ruft man das Skript in der Kommandozeile `"python python_lernen.py"` direkt auf, dann wird das Skript in Bytecode übersetzt und ausgeführt. Diesen Byte-Code bekommt die Anwenderin oder der Anwender nicht zu Gesicht.

Das ändert sich, wenn man in einem anderen Python-Skript oder auch in der Python-Shell dieses Skript mittels dem Kommando `"import"`, also `"import python_lernen"` importiert. Dann wird zusätzlich im gleichen Verzeichnis, in dem sich die Datei `python_lernen.py` befindet, eine Datei gleichen Namens aber mit der Endung `.pyc` angelegt, in der sich der Byte-Code befindet.



```
monty@python $ python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:58:18)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
⇒ import python_lernen
Python lernen!
⇒
monty@python $ ls python_lernen*
python_lernen.py python_lernen.pyc
monty@python $
```


Bei einem späteren import in einem anderen Programmlauf, wird dann direkt die Datei mit dem Byte-Code geladen. Importiert man die gleiche Datei mehrmals im gleichen Skript wird sie nur beim ersten Mal geladen.

Bei dem Byte-Code handelt es sich um einen Maschinenunabhängigen Code, der mittels einer virtuellen Maschine ausgeführt (PVM, Python Virtual Machine) wird.

2.2.5 Compiler

Ein Compiler (auch Übersetzer genannt) ist ein Computerprogramm, das ein in einer Quellsprache, wie beispielsweise C oder C++, geschriebenes Programm - Quelle oder Quellprogramm genannt - in ein semantisch äquivalentes Programm einer Zielsprache (Zielprogramm) übersetzt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes in Assemblersprache, Bytecode oder Maschinensprache. Das Übersetzen eines Quellprogramms in ein Zielprogramm durch einen Compiler wird auch als Kompilierung bezeichnet.

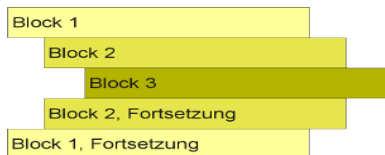
2.2.6 Interpreter

Ein Interpreter ist ein Programm, das einen Quellcode im Gegensatz zu Assemblern oder Compilern nicht direkt in ausführbaren Code, also eine ausführbare Datei, wandelt, sondern den Quellcode einliest, analysiert und ausführt. Die Analyse des Quellcodes erfolgt zur Laufzeit des Programms.

2.2.7 Strukturierung durch Einrückung

Das Strukturierungsprinzip von Python unterscheidet sich deutlich von anderen Programmiersprachen. Andere Sprachen strukturieren (klammern) Programmblöcke durch Schlüsselwörter, wie beispielsweise "begin", "end", "do", "done" oder geschweiften Klammern. Leerzeichen, Folgen von Leerzeichen oder Einrückungen sind für die Compiler und Interpreter von den meisten Programmiersprachen ohne jede Semantik, d.h. sie werden überlesen. Dennoch wird Programmierern aber immer empfohlen, Blöcke durch gleichmäßige Einrückungen für menschliche Benutzer kenntlich zu machen. In Python ist dies nun gänzlich anders. Hier haben Leerzeichen eine Bedeutung. Die Einrückung von Zeilen dient hier als Strukturierungselement, so dass Programmierer "gezwungen" werden übersichtlichen Code zu schreiben.

Ein häufiges Strukturierungselement sind Anweisungen die sich aus einem Anweisungskopf und einem Anweisungskörper zusammensetzen, wie z.B. die while- und die for-Schleife.



Wesentlich sind hierbei der Doppelpunkt am Ende des Anweisungskopfes und die gleichmäßige

Einrückung der zugehörigen Anweisungen.

2.2.8 Datentypen und Variablen

Auch wenn man Variablen und Datentypen von anderen Programmiersprachen bereits zur Genüge zu kennen glaubt, sollte man dieses Kapitel dennoch zumindest überfliegen, denn einiges ist eben doch anders in Python.

Eine Variable im allgemeinsten Sinne ist einfach ein Behälter (Container) zur Aufbewahrung von bestimmten Werten, also z.B. Strings oder Zahlen. Man kann im Verlauf des Programms auf diese Variablen, oder genauer auf den Wert ihres Inhaltes zugreifen, oder ihnen einen neuen Wert zuweisen. In den meisten Programmiersprachen, wie z.B. C, ist es so, dass eine Variable einen festen Speicherplatz bezeichnet, in dem Werte eines bestimmten Datentyps abgelegt werden können. Während des Programmlaufes kann sich der Wert der Variable ändern, aber die Wertänderungen müssen vom gleichen Typ sein. Also man kann nicht in einer Variable zu einem bestimmten Zeitpunkt eine Integerzahl gespeichert haben und dann diesen Wert durch eine Fließkommazahl überschreiben. Ebenso ist der Speicherort der Variablen während des gesamten Laufes konstant, kann also nicht mehr geändert werden. In Sprachen wie C wird der Speicherort bereit durch den Compiler fixiert. In Python sieht dies anders aus. Zunächst einmal bezeichnen Variablen in Python keinen bestimmten Typ und deshalb benötigt man auch in Python keine Typdeklaration. Benötigt man im Programm beispielsweise eine Variable `i` mit dem Wert 42, so erreicht man dies einfach mit der folgenden Anweisung:

```
i = 42
```

Obige Anweisung darf man nicht als mathematisches Gleichheitszeichen sehen, sondern als "der Variablen `i` wird der Wert 42 zugewiesen", d.h. der Inhalt von `i` ist nach der Zuweisung 42. Man kann diesen Wert der Variablen auch, wie man im folgenden Beispiel sieht, anschließend ändern:

```
⇒ i = 42
⇒ i = i + 1
⇒ print i
43
⇒
```

2.2.9 Zahlen

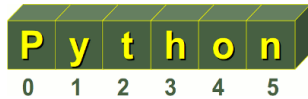
Python kennt vier eingebaute (built-in) Datentypen für Zahlen:

- Ganzzahl (Integer) z.B. 4321 vorangestellte 0 bedeutet Oktalzahl und vorangestelltes 0x bedeutet Hexzahl
- lange Ganzzahl Sie können beliebig lang werden Sie werden mit einem l am Anfang bzw. L am Ende bezeichnet.
- Fließkommazahlen Zahlen der Form 3.14159 oder 17.3e+02

- komplexe Zahlen z.B. $1.2+3j$

2.2.10 Zeichenketten - Strings

Ein String, oder Zeichenkette, kann man als eine Sequenz von einzelnen Zeichen sehen.



Jedes einzelne Zeichen eines Strings, kann über einen Index angesprochen werden. Im folgenden Beispiel sehen wir, wie der obige im Bild dargestellte String in Python definiert wird und wie wir auf ihn zugreifen können:

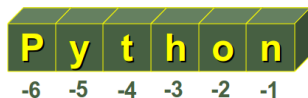
```
⇒ s = "Python"
⇒ print s[0]
P
⇒ print s[3]
h
```

Die Länge eines Strings kann man mit der Funktion `len()` bestimmen und damit kann man auch einfach beispielsweise auf das letzte oder vorletzte Zeichen eines Strings zugreifen:

```
⇒ s = "Python"
⇒ index_last_char = len(s) - 1
⇒ print s[index_last_char]
n
⇒ print s[index_last_char - 1]
o
⇒
```

Der Indexzähler beginnt immer bei 0 an aufwärts zu zählen.

Da es bei der praktischen Arbeit sehr häufig vorkommt, dass man auf einzelne Zeichen eines Strings von hinten zugreifen muss, wäre es sehr lästig, wenn man dies immer über den Umweg durch den Aufruf der Funktion `len()` machen müsste. Python bietet deshalb eine elegantere Möglichkeiten. Die Indices werden auch von rechts durch negative Indices nummeriert, d.h. das letzte Zeichen wird mittels des Index -1, das vorletzte mittels -2 usw. angesprochen.



Strings können unter Benutzung von

- **Konkatenation/Verbinden** (englisch: Concatenation)
Diese Funktion dient dazu mittels des "+"-Operators zwei Strings zu einem neuen String

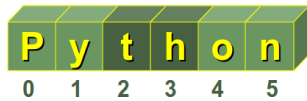
zusammenzuhängen: "Hello" + "World" ⇒ "HelloWorld"

- **Wiederholung** (englisch: Repetition)
Ein String kann wiederholt konkateniert werden. Dazu benutzt man den '*'-Operator. Beispiel:
"*_*" * 3 wird zu "*_*_*_*_*"

- **Indexing**
"Python"[0] ⇒ "P"

- **Slicing**
Das englische Verb "to slice" bedeutet in Deutsch "schneiden" oder auch "in Scheiben schneiden". Letztere Bedeutung entspricht auch der Funktion von Slicing in Python. Man schneidet sich gewissermaßen eine "Scheibe" aus einem String heraus. [2:4] bedeutet im folgenden Ausdruck, dass wir aus dem String "Python" einen Teilstring herauserschneiden, der mit dem Zeichen des Index 2 (inklusive) beginnt und bis zum Index 4 (exklusive) geht:

"Python"[2 : 4] ⇒ "th"



- **Länge** eines Strings
len("Python") ⇒ 6

angegeben werden.

2.2.11 Unveränderliche Zeichenketten

Wie in Java aber nicht wie in C oder C++, können Strings in Python nicht verändert werden. Versucht man eine indizierte Position zu ändern, erzeugt man eine Fehlermeldung:

```
⇒ s = "Some things are immutable!"
⇒ s[-1] = "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
⇒
```

2.2.12 Escape- oder Fluchtzeichen

Es gibt Zeichenfolgen, die den Textfluss steuern, wie zum Beispiel ein Newline (Zeilenvorschub) oder Tabulator. Sie lassen sich nicht auf dem Bildschirm als einzelne Zeichen darstellen. Die Darstellung solcher Zeichen innerhalb von String-Literalen erfolgt mittels spezieller Zeichenfolgen, sogenannter Escape-Sequenzen. Eine Escape-Sequenz wird von einem Backslash eingeleitet, gefolgt von der

Kennung des gewünschten Sonderzeichens. Übersicht der Escape-Zeichen:

- `\` Zeilenfortsetzung
- `\\` Rückwärtsschrägstrich
- `\'` Einzelnes Anführungszeichen
- `\"` Doppeltes Anführungszeichen
- `\a` Glocke
- `\b` Rückschritt
- `\e` Ausmaskieren
- `\0` Null
- `\n` Zeilenvorschub (linefeed, LF)
- `\v` Vertikaler Tabulator
- `\t` Horizontaler Tabulator
- `\r` Wagenrücklauf (carriage return, CR)
- `\f` Seitenvorschub
- `\0XX` Oktaler Wert
- `\xXX` Hexadezimaler Wert

Die Auswertung von Escape-Zeichen kann man verhindern, indem man einem String ein `r` oder `R` unmittelbar voranstellt.

Beispiel:

`r"\n` bewirkt einen Zeilenvorschub"

2.2.13 Typwechsel bei Variablen

In Python kann eine Variable, wie bereits gesagt, sofort ohne Deklaration des Datentyps verwendet werden. Dennoch vergibt Python einen Datentyp, d.h. je nach Datentyp, wird die Variable anders angelegt, also als Integer, Float, String, und so weiter. Eigentlich müsste man sagen, dass eine Objekt mit einem bestimmten Datentyp bzw. Klasse angelegt wird. Die Variable referenziert dann dieses Objekt, d.h. die Variable selbst hat also eigentlich keinen Typ. Anders ausgedrückt Der Datentyp ist in Python nicht an die Variable, sondern an den Wert gebunden, was impliziert, dass sich der Typ zur Laufzeit ändern kann, wie wir im folgenden Beispiel sehen können:

```
i = 42          # Datentyp ist integer (implizit)
i = 42 + 0.11   # Typ ändert sich zu float
i = "fourty"    # und jetzt ein String
```

2.2.14 Wechselnde Speicherorte

Prinzipiell wird sich im vorigen Fall, wobei das natürlich implementierungsabhängig ist, der Speicherort für die Variable `i` ändern. Der Interpreter kann bei der Anweisung `"i = 42"` den Wert als Integer abspeichern, muss aber bei der Anweisung `"i = 42 + 0.11"` einen neuen Ort für eine Float-Zahl anlegen. Für `i = "fourty"` muss er in einen String gewandelt werden. Achtung: Als Anwender braucht man dies eigentlich nicht zu wissen, da ja alles automatisch geschieht!

Betrachten wir nun folgenden Python-Code:

```
⇒ x = 3
⇒ y = x
⇒ y = 2
```

Intuitiv würde man davon ausgehen, dass Python zunächst für `x` einen Speicherort wählt und dort das Objekt (Zahl) 3 abspeichert. Anschließend wird der Variablen `y` der Wert von `x` zugewiesen. In C und vielen anderen Programmiersprachen würde auch für `y` ein eigener Speicherort bestehen, in dem nun die Zahl 3 hineingeschrieben würde. Python geht anders vor: `x` ist eine Variable mit dem Objekt 3 und `y` ist eine Variable mit dem "selben" (nicht "gleichen") Objekt. `x` und `y` "zeigen" auf das gleiche Objekt. In der letzten Zeile wird `y` nun der Wert 2 zugewiesen, jetzt muss ein neues Objekt angelegt werden und `y` "zeigt" auf einen neuen Speicherort. (Anm: Dieses eben verwendete 'zeigen' sollte von C-Programmierern keinesfalls mit den unter C verwendeten Pointern verwechselt werden.)

Es stellt sich nun die Frage, wie man das oben gesagte überprüfen kann. Dazu bietet sich die Identitätsfunktion `id()` an. Die Identität einer Instanz dient dazu, sie von allen anderen Instanzen zu unterscheiden. Die Identität ist eine Ganzzahl, und sie ist innerhalb eines Programmes eindeutig. Die Identitätsfunktion `id()` liefert die Identität. So kann man prüfen, ob es sich um eine bestimmte Instanz handelt und nicht nur um eine mit dem gleichen Wert und Typ. Wir geben nochmals das obige Beispiel ein, lassen uns aber jeweils die Identität ausgeben:

```
⇒ x = 3
⇒ print id(x)
157379912
⇒ y = x
⇒ print id(y)
157379912
⇒ y = 2
⇒ print id(y)
157379924
⇒ print id(x)
157379912
⇒
```

Wir stellen fest, dass sich die Identität erst ändert, nachdem wir `y` einen neuen Wert zugewiesen haben. Die Identität von `x` bleibt gleich, d.h. der Speicherort von `x` wird nicht geändert.

2.2.15 Besonderheiten bei Strings

Einen besonderen Effekt können wir bei Strings feststellen. Im folgenden Beispiel wollen wir dies veranschaulichen. Dazu benötigen wir noch den 'is'-Operator. Sind a und b zwei Strings, dann prüft "a is b", ob a und b die gleiche Identität (Speicherort) haben. Wenn 'a is b' gilt, dann gilt auch "a == b". Aber wenn "a == b" gilt, dann gilt natürlich nicht notwendigerweise auch "a is b"! Nun wollen wir untersuchen, wie Strings in Python abgespeichert werden. Im folgenden Beispiel, erkennen wir, dass a und b sich den gleichen Speicherort teilen, obwohl wir diesmal keine Zuweisung der Art "b = a" verwendet haben:

```
⇒ a = "Linux"
⇒ b = "Linux"
⇒ a is b
True

⇒ a = "Baden-Württemberg"
⇒ b = "Baden-Württemberg"
⇒ a is b
False
⇒ a == b
True
```

2.3 Ausdrücke und Operatoren

Unter einem Ausdruck in Python und in anderen Programmiersprachen versteht man eine Kombination aus Variablen, Konstanten, Operatoren und Rückgabewerten von Funktionen. Die Auswertung eines Ausdrucks ergibt einen Wert, der meistens einer Variablen zugewiesen wird. In Python werden Ausdrücke unter Verwendung der gebräuchlichen mathematischen Notationen und Symbolen für Operatoren geschrieben.

2.3.1 Operatoren

Die meisten Operatoren für Zahlenwerte sind in Python ähnlich zu anderen Programmiersprachen. Wir geben hier eine Übersicht, ohne sie vollständig zu erklären. Bei Bedarf werden diese Operatoren in anderen Kapitel besprochen.

Operator	Bezeichnung	Beispiel
+, -	Addition, Subtraktion	10 - 3
*, /, %	Multiplikation, Division, Rest	
+x, -x	Vorzeichen	-3
~x	Bitweises Not	~3 - 4
or, and, not	Boolsches Oder, Und, Nicht	(a or b) and c
in	"Element von"	1 in [3, 2, 1]
<, <=, >, >=, !=, ==	Vergleichsoperatoren	2 <= 3
~, &, ^	Bitweises Oder, Und, XOR	6 ^ 3
<<, >>	Shiftoperatoren	6 << 2

2.4 Tiefes und flaches Kopieren

Wie wir im letzten Kapitel "Datentypen und Variablen" gesehen haben, verhält sich Python beim Kopieren einfacher Datentypen wie Integer und Strings ungewöhnlich im Vergleich zu anderen Programmiersprachen. In dem folgenden Code-Beispiel zeigt y zunächst nur auf den gleichen Speicherplatz wie x. Erst wenn der Wert von y verändert wird, erhält y einen eigenen Speicherplatz, wie wir im vorigen Kapitel gesehen haben.

```
⇒ x = 3
⇒ y = x
```

Aber auch wenn das obige Verhalten ungewöhnlich im Vergleich zu anderen Programmiersprachen wie C, C++, Perl und anderen ist, so entsprechen die Resultate der Zuweisungen dennoch unseren Erwartungen. Kritisch wird es jedoch, wenn wir mutable Objekte wie Listen und Dictionaries kopieren wollen. Python legt nur dann echte Kopien an, wenn es unbedingt muss, d.h. dass es der Anwender, also der Programmierer, explizit verlangt. In diesem Kapitel wollen wir einige Probleme aufzeigen, die beim Kopieren von mutablen Objekten entstehen können, also z.B. beim kopieren von Listen und Dictionaries.

2.4.1 Kopieren einer Liste

```
⇒ colours1 = [ "red", "green" ]
⇒ colours2 = colours1
⇒ colours2 = [ "rouge", "vert" ]
⇒ print colours1
[ 'red', 'green' ]
```

In dem obigen Code-Beispiel legen wir zuerst eine einfache Liste colours1 an, die wir dann in colours2 kopieren. Anschließend weisen wir colours2 eine neue Liste zu.

Es erstaunt wenig, dass die Werte von colours1 dabei unverändert bleiben. Wie bei dem Beispiel mit den Integer-Variablen im letzten Kapitel "Datentypen und Variablen" wird ein neuer Speicherbereich für colours2 angelegt, als dieser Variablen eine komplett neue Liste zugeordnet wird.


```

=> colours1 = ["red", "green"]
=> colours2 = colours1
=> colours2[1] = "blue"
=> colours1
['red', 'blue']

```

Wie sieht es jedoch aus, wenn nur einzelne Elemente geändert werden? Um dies zu testen weisen wir dem zweiten Element von `colours2` einen neuen Wert zu. Viele wird es nun erstauen, dass auch `colours1` damit verändert wurde, obwohl man doch eine Kopie von `colours1` gemacht zu haben glaubte. Die Erklärung ist, dass das zugehörige Objekt von `colours2` nicht geändert wurde.

Mit dem Teilbereichsoperator (slicing) kann man flache Listenstrukturen komplett kopieren, ohne dass es zu Seiteneffekten kommt, wie man im folgenden Beispiel sehen kann:

```

=> liste1 = ['a','b','c','d']
=> liste2 = liste1[:]
=> liste2[1] = 'x'
=> print liste2
['a', 'x', 'c', 'd']
=> print liste1
['a', 'b', 'c', 'd']
=>

```

Sobald jedoch auch Unterlisten in der zu kopierenden Liste vorkommen, werden nur Zeiger auf diese Unterlisten kopiert.

```

=> lst1 = ['a','b',['ab','ba']]
=> lst2 = lst1[:]

```

Weist man nun zum Beispiel dem 0-ten Element einer der beiden Listen einen neuen Wert zu, führt dies nicht zu einem Seiteneffekt. Probleme gibt es erst, wenn man direkt eines der beiden Elemente der Unterliste verändert. Um dies zu demonstrieren, ändern wir nun zwei Einträge in `lst2`:

```

=> lst1 = ['a','b',['ab','ba']]
=> lst2 = lst1[:]
=> lst2[0] = 'c'
=> lst2[2][1] = 'd'
=> print(lst1)
['a', 'b', ['ab', 'd']]
=>

```

Man erkennt, dass man aber nicht nur die Einträge in `lst2` geändert hat, sondern auch den Eintrag von `lst1[2][1]`. Dies liegt daran, dass in beiden Listen, also `lst1` und `lst2`, das jeweils dritte Element nur ein Link auf eine physikalisch gleiche Unterliste ist. Diese Unterliste wurde nicht mit `[:]` mitkopiert.

2.4.2 Kopie mit der Methode deepcopy aus dem Modul copy

Abhilfe für das eben beschriebene Problem schafft das Modul "copy". Dieses Modul stellt die Methode "deepcopy" zur Verfügung, die das komplette Kopieren einer nicht flachen Listenstruktur erlaubt.

Das folgende Skript kopiert unser obiges Beispiel nun mit Hilfe dieser Methode:

```
from copy import deepcopy

lst1 = ['a','b',['ab','ba']]

lst2 = deepcopy(lst1)

lst2[2][1] = "d"
lst2[0] = "c";

print lst2
print lst1
```

Speichern wir das Skript unter deep_copy.py ab und rufen es mit "python deep_copy.py" auf, so erhalten wir folgende Ausgabe:

```
$ python deep_copy.py ['c', 'b', ['ab', 'd']]
['a', 'b', ['ab', 'ba']]
```

2.5 Bedingte Anweisungen

Zu bestimmten Zeiten sind gewisse Entscheidungen unvermeidlich, wie man im Foto sehen kann. So kann man kaum ein sinnvolles Programm schreiben, welches ohne jede Verzweigung auskommt. Bisher sind wir in der Lage Programme zu schreiben, in denen eine Anweisung auf die andere folgt und diese auch in dieser Reihenfolge ausgeführt werden. Aber nur wenige Probleme lassen sich durch einen linearen Programmablauf kontrollieren. Man möchte beispielsweise einen bestimmten Teil des Programmes nur dann ausführen, wenn bestimmte Bedingungen zutreffen oder ein anderer Teil soll gegebenenfalls mehrmals ausgeführt werden. Dazu bietet jede Programmiersprache Kontrollstrukturen, die sich in zwei Kategorien unterscheiden lassen: Verzweigungen und Schleifen.

In einer Programmiersprache versteht man unter einer bedingten Anweisung, wie man Verzweigungen auch nennt, Codeteile, die unter bestimmten Bedingungen ausgeführt werden. Liegt die Bedingung nicht vor, wird dieser Code nicht ausgeführt. Anders ausgedrückt: Eine Verzweigung legt fest welcher von zwei (oder auch mehr) Programmteilen (Alternativen) in Abhängigkeit von einer (oder mehreren) Bedingungen ausgeführt wird. Bedingte Anweisungen und Verzweigungen werden in Programmiersprachen (ebenso wie die Schleifen) den Kontrollstrukturen zugerechnet, weil mit ihrer Hilfe ein Programm auf verschiedene Zustände, die sich aus Eingaben und Berechnungen ergeben, reagieren kann.

In dem abschließenden Beispiel dieses Kapitels geht es um Steuern. Wir schreiben ein Python-

skript, mit dem man sich aus dem zu versteuernden Einkommen die Steuern für 2010 berechnen lassen kann.

2.5.1 Die if-Anweisung

Die allgemeine Form der if-Anweisung sieht in Python wie folgt aus:

```
if bedingung1:
    anweisungen1
elif bedingung2:
    anweisungen2
else:
    anweisungen3
```

Falls die Bedingung "bedingung1" wahr ist, werden die Anweisungen "anweisungen1" ausgeführt. Wenn nicht, werden, falls bedingung2 wahr ist, die anweisungen2 ausgeführt. Falls weder die erste Bedingung (bedingung1) noch die zweite Bedingung (bedingung2) wahr ist, werden die Anweisungen nach dem else (anweisungen3) ausgeführt.

2.5.2 Beispiel Hundejahre

Kinder und Hundeliebhaber stellen sich häufig die Frage, wie alt ihr Hund wohl wäre, wenn er kein Hund sondern ein Mensch wäre. Landläufig rechnet man Hundejahre in Menschjahre um, indem man das Alter des Hundes mit 7 multipliziert. Je nach Hundegröße und Rasse sieht die Umrechnung jedoch etwas komplizierter aus, z.B.:

```
age = input("Alter des Hundes: ")
print
if age < 0:
    print "Das stimmt wohl kaum!"
elif age == 1:
    print "entspricht ca. 14 Jahre"
elif age == 2:
    print "entspricht ca. 22 Jahre"
elif age > 2:
    human = 22 + (age - 2) * 5
    print "Menschenjahre", human

###
raw_input('press Return>')
```

2.5.3 Wahr oder falsch

Leider ist es nicht in allen Dingen des Lebens so einfach zwischen Wahr und Falsch zu unterscheiden wie in Python: Als "falsch" gelten

- numerische Null-Werte(0, 0L, 0.0, 0.0+0.0j),
- der booleschen Wert False,
- leere Zeichenketten,
- leere Listen, leere Tupel,
- leere Dictionaries.
- sowie der spezielle Wert None.

Alle anderen Werte betrachtet Python als "wahr".

2.5.4 Abgekürztes IF

C-Programmierer kennen folgende abgekürzte Schreibweise für das if-Konstrukt:

```
max = (a > b) ? a : b;
```

als Abkürzung für:

```
if (a > b)
max=a;
else
max=b;
```

In Python müssen sich C-Programmierer, was diese Schreibweise betrifft, umgewöhnen. In Python formuliert man das vorige Beispiel wie folgt:

```
max = a if (a > b) else b;
```

2.5.5 Steuerrechner in Python

Das zu versteuernde Einkommen (zvE) wird zunächst einer Tarifzone zugeordnet. Dann lässt sich der Steuerbetrag (StB) bei Einzelveranlagung nach der entsprechenden Formel berechnen.

- Erste Zone (Grundfreibetrag): bis 8004,- € fällt keine Steuer an
- Zweite Zone: zvE von 8.005 € bis 13.469 €

$$\text{StB} = (912,17 * y + 1400) * y$$
Für y gilt:

$$y = (\text{zvE} - 8004) / 10000$$

- Dritte Zone: zvE von 13470 € bis 52881 €
 $\text{StB} = (228,74 * z + 2397) * z$
Für z gilt:
 $z = (\text{zvE} - 13469) / 10000$
- Vierte Zone: zvE von 52882 € bis 250730 €
 $\text{StB} = 0,42 * \text{zvE} - 8172$
- Fünfte Zone: zvE ab 250731 €
 $\text{StB} = 0,44 * \text{zvE} - 15694$

Eine Python-Funktion zur Berechnung der Steuer aus dem Einkommensteuertarif sieht nun wie folgt aus: (natürlich ohne Gewähr!!!)

```
def steuern(einkommen):
    """Berechnung der zu zahlenden Steuern fuer ein zu versteuerndes Einkommen von x"""
    if einkommen <= 8004:
        steuer = 0
    elif einkommen <= 13469:
        y = (einkommen - 8004.0) / 10000.0
        steuer = (912.17 * y + 1400) * y
    elif einkommen <= 52881:
        z = (einkommen - 13469.0) / 10000.0
        steuer = (228.74 * z + 2397.0) * z + 1038.0
    elif einkommen <= 250730:
        steuer = einkommen * 0.42 - 8172.0
    else:
        steuer = einkommen * 0.44 - 15694
    return steuer
```

2.6 Eingabe mit input()

Es gibt kaum Programme, die ohne jegliche Eingaben auskommen. Eingaben können über viele Wege erfolgen, so zum Beispiel aus einer Datenbank, von einem anderen Rechner im lokalen Netzwerk oder auch über das Internet. Die einfachste und wohl auch häufigste Eingabe erfolgt jedoch über die Tastatur. Für diese Form der Eingabe bietet Python die Funktion `input(text)`.

Kommt es zum Aufruf der Funktion `input` während eines Programmlaufes, wird der Programmablauf solange gestoppt, bis die Benutzerin oder der Benutzer eine Eingabe über die Tastatur tätigt und diese mit der Return-Taste abschließt. Damit der User auch weiß, was er einzugeben hat, wird der String des Parameters `text` ausgegeben, sofern ein solcher String existiert. Der Parameter von `input()` ist optional.

Der Eingabestring des Benutzers wird von `input()` interpretiert, d.h. `input()` liefert beispielsweise einen Integer Wert zurück, wenn der Benutzer eine ganze Zahl eingeben hat und eine Liste, wenn der Benutzer eine Liste eingegeben hat.

Wir zeigen dies in der folgenden interaktiven Python-Shell-Sitzung:

```
⇒ x = input("Ihr Name? ")
Ihr Name? "John"
⇒ print(x)
John
⇒ x = input("Ihre Gehalt? ")
Ihre Gehalt? 2877.03
⇒ x
2877.03
⇒ type(x)
<type 'float'>
⇒ x = input("Ihre Lieblingssprachen? ")
Ihre Lieblingssprachen? [ "Java", "Perl", "C++" ]
⇒ print(x)
['Java', 'Perl', 'C++']
⇒ type(x)
<type 'list'>
⇒
```

2.7 Eingabe mit `raw_input()`

Anders als bei `input`, interpretiert `raw_input` nicht die Eingabe. Das heißt, `raw_input` liefert immer einen String zurück, d.h. der Eingabestring des Benutzers wird unverändert weitergeleitet. Will man einen bestimmten Datentyp, so kann man die Eingabe durch die entsprechende Casting-Funktion wandeln oder man kann die `eval`-Funktion verwenden. Wir demonstrieren dies wieder an einigen Beispielen in der interaktiven Python-Shell:

```

=> x = raw_input("Ihr Name? ")
Ihr Name? John
=> print(x)
John
=> x = raw_input("Ihre Gehalt? ")
Ihre Gehalt? 2877.03
=> print(x)
2877.03
=> type(x)
<type 'str'>
=> x = float(raw_input("Ihre Gehalt? "))
Ihre Gehalt? 2877.03
=> type(x)
<type 'float'>
=> x = eval(raw_input("Ihre Lieblingssprachen? "))
Ihre Lieblingssprachen? ["Java", "Perl", "C++"]
=> print(x)
['Java', 'Perl', 'C++']
=> type(x)
<type 'list'>
=>

```

2.8 Schleifen

Schleifen, werden benötigt, um einen Codeblock, den man auch als Schleifenkörper bezeichnet, wiederholt auszuführen. In Python gibt es zwei Schleifentypen: die while-Schleife und die for-Schleife. Die meisten Schleifen enthalten einen Zähler oder ganz allgemein Variablen, die im Verlauf der Berechnungen innerhalb des Schleifenkörpers ihre Werte ändern. Außerhalb, d.h. noch vor dem Beginn der Schleife, werden diese Variablen initialisiert. Vor jedem Schleifendurchlauf wird geprüft, ob ein Ausdruck, in dem diese Variable oder Variablen vorkommen, wahr ist. Dieser Ausdruck bestimmt das Endkriterium der Schleife. Solange die Berechnung dieses Ausdrucks True liefert wird der Rumpf der Schleife ausgeführt. Nachdem alle Anweisungen des Schleifenkörpers durchgeführt worden sind, springt die Programmsteuerung automatisch zum Anfang der Schleife, also zur Prüfung des Endkriteriums zurück und prüft wieder, ob diese nochmals erfüllt ist. Wenn ja, geht es wie oben beschrieben weiter, ansonsten wird der Schleifenkörper nicht mehr ausgeführt und es wird mit dem Rest des Skriptes fortgefahren. Das nebenstehende Diagramm zeigt dies schematisch.

Wir möchten dies nun an einem kleinen Python-Skript verdeutlichen, welches die Summe der Zahlen von 1 bis 100 bildet. Im nächsten Kapitel über for-Schleifen werden wir eine weitaus elegantere Möglichkeiten zu diesem Zweck kennenlernen.

```
n = 100

s = 0
i = 1
while i <= n:
    s = s + i
    i = i + 1

print "Die Summe lautet: ", s
```

2.8.1 Standard-Eingabe lesen

Bevor wir mit der while-Schleife weitermachen, müssen wir noch ein paar grundsätzliche Dinge über die Standardeingabe und die Standardausgabe klären. Als Standardeingabe gilt normalerweise die Tastatur. Die meisten Shell-Programme schreiben ihre Ausgaben in die Standardausgabe, d.h. das Terminalfenster oder die Textkonsole. Fehlermeldungen werden in die Standard-Fehlerausgabe ausgegeben, was üblicherweise auch dem aktuellen Terminalfenster oder der Textkonsole entspricht. Auch der Python-Interpreter stellt drei Standard-Dateiobjekte zur Verfügung:

- Standardeingabe
- Standardausgabe
- Standardfehlerausgabe

Sie stehen im Modul "sys" als:

- sys.stdin
- sys.stdout
- sys.stderr

zur Verfügung.

Das folgende Beispiel-Skript zeigt nun, wie man Zeichen für Zeichen mittels einer while-Schleife von der Standardeingabe (Tastatur) einliest. Mit dem import-Befehl wird das benötigte Modul sys eingelesen.


```
import sys

text = ""
while 1:
    c = sys.stdin.read(1)
    text = text + c
    if c == '\n':
        break

print "Eingabe: %s" % text
```

Eleganter kann man eine beliebige Eingabezeile von der Standardeingabe natürlich mit der Funktion `raw_input(prompt)` einlesen.

```
⇒ name = raw_input("Wie heißen Sie?setminus")
Wie heißen Sie?
Tux
⇒ print name
Tux
⇒
```

2.8.2 Der else-Teil

Wie auch die bedingte `if`-Anweisung hat die `while`-Schleife in Python im Gegensatz zu anderen Programmiersprachen einen optionalen `else`-Zweig, was für viele Programmierer gewöhnungsbedürftig ist. Die Anweisungen im `else`-Teil werden ausgeführt, sobald die Bedingung nicht mehr erfüllt ist. Sicherlich fragen sich einige nun, worin dann der Unterschied zu einer normalen `while`-Schleife liegt. Hätte man die Anweisungen nicht in den `else`-Teil gesteckt sondern einfach hinter die `while`-Schleife gestellt, wären sie ja auch genauso ausgeführt worden. Es wird erst mit einem `break`-Kommando, was wir später kennenlernen sinnvoll. Allgemein sieht eine `while`-Schleife mit `else`-Teil in Python wie folgt aus:

```
while Bedingung:
    Anweisung1
    ...
    Anweisungn
else:
    Anweisung1
    ...
    Anweisungn
```

2.8.3 Vorzeitiger Abbruch einer while-Schleife

Normalerweise wird eine Schleife nur beendet, wenn die Bedingung im Schleifenkopf nicht mehr erfüllt ist. Mit `break` kann man aber eine Schleife vorzeitig verlassen und mit `continue` einen Durch-

lauf beenden. Im folgenden Beispiel, einem einfachen Zahlenratespiel, kann man erkennen, dass in Kombination mit einem `break` der `else`-Zweig durchaus sinnvoll sein kann. Nur wenn die `while`-Schleife regulär beendet wird, d.h. der Spieler die Zahl erraten hat, gibt es einen Glückwunsch. Gibt der Spieler auf, d.h. `break`, dann wird der `else`-Zweig der `while`-Schleife nicht ausgeführt.

```
import random
n = 20
to_be_guessed = int(n * random.random()) + 1
guess = 0
while guess != to_be_guessed:
    guess = input("Neue Zahl: ")
    if guess > 0:
        if guess > to_be_guessed:
            print "Zahl zu groß"
        elif guess < to_be_guessed:
            print "Zahl zu klein"
    else:
        print "Schade, Sie geben also auf!"
        break
else:
    print "Glückwunsch! Das war's!"
```

2.9 for-Schleifen

Syntax:

```
for Variable in Sequenz:
    Anweisung1
    Anweisung2
    ...
    Anweisungn
else:
    Else-Anweisung1
    Else-Anweisung2
    ...
    Else-Anweisungm
```

Die `for`-Anweisung hat einen unterschiedlichen Charakter zu den `for`-Schleifen, die man aus den meisten anderen Programmiersprachen kennt. In Python dient die `for`-Schleife zur Iteration über eine Sequenz von Objekten, während sie in anderen Sprachen meist nur eine etwas andere `while`-Schleife ist.

Beispiel einer `for`-Schleife in Python:

```

=> languages = ["C", "C++", "Perl", "Python"]
=> for x in languages:
...     print x
...
C
C++
Perl
Python
=>

```

2.10 Die range()-Funktion

Mit Hilfe der range()-Funktion lässt sich die for-Schleife ideal für Iterationen nutzen. range() liefert Listen, die arithmetischen Aufzählungen entsprechen. Beispiel:

```

=> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

Obiges Beispiel zeigt, dass Range mit einem Argument aufgerufen die Liste der Zahlen von 0 bis zu diesem Argument liefert. range() kann aber auch mit zwei Argumenten aufgerufen werden:

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

Dann wird eine Liste aller ganzen Zahlen von begin (einschließlich) bis end (ausschließlich) geliefert. Als drittes Argument kann man range() noch die Schrittweite mitgeben. Beispiel:

```

=> range(4,10)
[4, 5, 6, 7, 8, 9]
=> range(4,50,5)
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]

```

Besonders sinnvoll wird die range()-Funktion im Zusammenspiel mit der for-Schleife. Im nachfolgenden Beispiel bilden wir die Summe der Zahlen von 1 bis 100:

```

n = 100
s = 0
for i in range(1, n+1):
    s = s + i
print s

```

In obigem kleinen Programm verbirgt sich aber noch ein schreckliches Effizienzproblem. Was geschieht bevor die for-Schleife ausgeführt wird? Python wertet zuerst den Aufruf range(1, n+1) aus. Das bedeutet, dass eine Liste mit 100 Zahlen erzeugt wird, also [1, 2, 3, 4, ... 100]. Es werden zwar alle Zahlen dieser Liste innerhalb der Schleife benötigt, aber zu keinem Zeitpunkt die ganze Liste. Im vorigen Kapitel hatte wir dieses Problem mit einer while-Schleife gelöst und dort benötigten wir auch keine Liste. Python bietet eine Lösung für dieses Problem, indem es die Funktion xrange zur Verfügung stellt. xrange erzeugt ein iterierbares Objekt (iterable), das bedeutet, dass keine Liste

erzeugt wird sondern zum Beispiel in einer for-Schleife über die Werte iteriert werden kann ohne dass die Liste erzeugt wird:

```

=> for i in xrange(1, 7):
...     print(i)
...
1
2
3
4
5
6
=>

```

Obige Schleife verhält sich im Hinblick auf die Effizienz ähnlich wie folgende while-Schleife:

```

=> i = 1
=> while i < 7:
...     print(i)
...     i += 1
...
1
2
3
4
5
6
=>

```

Im Ausgabeverhalten sieht man natürlich keinen Unterschied. Den Unterschied zwischen range und xrange sieht man aber, wenn man die Aufrufe direkt in der interaktiven Python-Shell tätigt:

```

=> range(1,7)
[1, 2, 3, 4, 5, 6]
=> xrange(1,7)
xrange(1, 7)
=>

```

Die meisten glauben, dass der Satz von Pythagoras von Pythagoras entdeckt worden war. Warum sonst sollte der Satz seinen Namen erhalten haben. Aber es gibt eine Debatte, ob dieser Satz nicht auch unabhängig von Pythagoras und vor allen Dingen bereits früher entdeckt worden sein könnte. Für die Pythagoräer - eine mystische Bewegung, die sich auf die Mathematik, Religion und die Philosophie begründete - waren die ganzen Zahlen, die den Satz des Pythagoras erfüllten besondere Zahlen, die für sie heilig waren.

Heutzutage haben die Pythagoräischen Zahlen nichts mystisches mehr. Obwohl sie für manche Schülerin oder Schüler oder anderer Personen, die mit der Mathematik auf Kriegsfuß stehen, immer noch so erscheinen mögen. Ganz unromantisch gilt in der Mathematik:

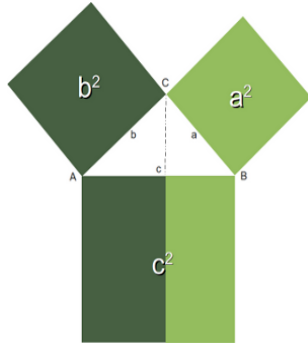


Abbildung 2.1: Pythagoras-Satz-Beweis

Drei natürliche Zahlen, welche die Gleichung $a^2 + b^2 = c^2$ erfüllen, heißen pythagoräische Zahlen. Das folgende Programm berechnet alle pythagoräischen Zahlen bis zu einer einzugebenden maximalen Zahl:

```
#!/usr/bin/env python
from math import sqrt
n = raw_input("Maximale Zahl? ")
n = int(n)+1
for a in xrange(1, n):
    for b in xrange(a, n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print a, b, c
```

2.11 Iteration über Liste mit range()

Falls man auf die Indexe einer Liste zugreifen möchte, scheint es keine gute Idee zu sein eine For-Schleife zur Iteration über die Liste zu nutzen. Man kann dann zwar alle Elemente erreichen, aber der Index eines Elementes ist nicht verfügbar. Aber es gibt eine Möglichkeit sowohl auf den Index als auch auf das Element zugreifen zu können. Die Lösung besteht darin range() in Kombination mit der len()-Funktion, die einem die Anzahl der Listenelemente liefert, zu benutzen:

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

2.12 Listen-Iteration mit Seiteneffekten

Falls man über eine Liste iteriert, sollte man vermeiden die Liste im Schleifenkörper (body) zu verändern. Was passieren kann, zeigen wir im folgenden Beispiel:

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

Was wird durch die Anweisung `print colours` ausgegeben?

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

Am besten benutzt man eine Kopie der Liste, wie im nächsten Beispiel:

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

Die Ausgabe sieht nun wie folgt aus:

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

Auch jetzt haben wir die Liste verändert, aber "bewusst" innerhalb des Schleifenkörpers. Aber der Elemente, die über die For-Schleife iteriert werden, bleiben unverändert durch die Iterationen.

2.13 Ausgabe mit print

Es gibt eigentlich keine Computer-Programme und natürlich auch keine Python-Programme, die nicht in irgendeiner Weise mit der Außenwelt kommunizieren. Vor allen Dingen muss ein Programm immer Ergebnisse ausgeben. Eine Form der Ausgabe von Ergebnissen geht direkt in die Standardausgabe und dies erfolgt in Python mittels der `print`-Anweisung.

```

=> print "Hallo Benutzer"
Hallo Benutzer
=> antwort = 42
=> print "Die Antwort lautet: " + str(antwort)
Die Antwort lautet: 42
=>

```

Ab Python3 ist print keine Anweisung mehr, sondern eine Funktion. Deshalb müssen die Argumente in Klammern geschrieben werden. Auch in Python2 kann man print mit Klammern schreiben:

```

=> print("Hallo")
Hallo
=> print("Hallo","Python")
('Hallo', 'Python')
=> print "Hallo","Python"
Hallo Python
=>

```

Wir können erkennen, dass es dann ein geändertes Ausgabeverhalten gibt. Aber was noch schwerwiegender ist. Das Ausgabeverhalten mit runden Klammern in Python 2.x ist auch anders als das der Python-Funktion von Version 3.x, wie wir im Folgenden sehen können:

```

$ python3
Python 3.2.3 (default, Apr 10 2013, 05:03:36)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
=> print("Hallo")
Hallo
=> print("Hallo","Python")
Hallo Python
=>

```

2.13.1 Import aus der Zukunft: print_function

In manchen Python2-Programmen findet sich die folgende Import-Zeile:

```
from __future__ import print_function
```

Dies ist auf den ersten Blick missverständlich. Es sieht so aus, als habe man eine Funktion namens "print_function" importiert. Es wurde aber nur ein Flag, eine Instanz "print_function" gesetzt. Dadurch wird in Python2 die print-Funktion von Python3 eingeführt. Das bedeutet, dass dann der Interpreter nur noch die Python3-Syntax der Print-Funktion akzeptiert. Das Programm verhält sich damit, zumindest was die print-Funktion betrifft, kompatibel zu Python3.

Wenn Sie Programme in Python 2.6 oder Python 2.7 erstellen wollen, empfehlen wir Ihnen, diese immer mit obigem Import zu versehen. Dadurch ist sichergestellt, dass Ihre Ausgaben kompatibel zu Python 3 sind.

Grundlagen - Mathematik

3.1 Zahlensysteme

In jedem Zahlensystem wird der Wert einer Ziffer durch ihre Stellung innerhalb einer Zahl bestimmt. Die Grundlage eines jedem Zahlensystems ist seine Basis.

$$\begin{array}{rcl} \text{Beispiel: } m * b^n & m & = \text{ Ziffernwert} \\ & b & = \text{ Basis} \\ & n & = \text{ Exponent} \\ & b^n & = \text{ Zahlenbasis} \end{array}$$

$36_{(10)}$ Zahlenbasis : 10
Zehner und Einer benennen den Stellenwert.

Wählt man dagegen einen andere Zahlenbasis, so ändert sich der Wert beträchtlich.

$$\begin{array}{rcl} 36_{(10)} & \text{Zahlenbasis : 16} & \\ \text{Das ergibt:} & 3 * 16^1 = & 48 \\ & + 6 * 16^0 = & 6 \\ & = 6 * 16^0 = & 54 \end{array}$$

3.1.1 Dualsystem

Das duale System entspricht dem binären Aufbau der elektronischen Datenverarbeitung. Es beruht auf der Basis 2 und benötigt nur die Ziffern 0 und 1.

$$\begin{array}{rcl} \text{Beispiel: } 10_{(2)} & = & 1 * 2^1 = 2 \\ & + & 0 * 2^0 = 0 \\ & & = 2 \end{array}$$

3.2 Umwandlung Dezimal - Dual

Im ersten Fall wird die Dezimalzahl so lange durch die Basis 2 geteilt, bis das Ergebnis Null erreicht ist. Die Reste der einzelnen Divisionen ergeben die Dualzahl. Im zweiten Fall werden die Ziffernwerte mit den Stellenwerten der Dualzahl multipliziert. Die Summe der Produkte ergibt die Dezimalzahl.

3.2.1 Dezimal nach Dual umwandeln

$$\begin{array}{rcl} 29 : 2 & = & 14 + \text{Rest } 1 \\ 14 : 2 & = & 7 + \text{Rest } 0 \\ 7 : 2 & = & 3 + \text{Rest } 1 \\ 3 : 2 & = & 1 + \text{Rest } 1 \\ 1 : 2 & = & 0 + \text{Rest } 1 \end{array} = 11101_{(2)}$$

3.2.2 Dual in Dezimalzahl umwandeln

$$\begin{array}{rclclcl}
 1 & 1 & 1 & 0 & 1 & = & 1 * & 1 & = & 1 \\
 & & & & & & 0 * & 0 & = & 0 \\
 & & & & & & 1 * & 4 & = & 4 \\
 & & & & & & 1 * & 8 & = & 8 \\
 & & & & & & 1 * & 16 & = & 16 = 29_{(10)}
 \end{array}$$

3.3 Hexadezimalsystem

In diesem System liegt die Basis 16 zu Grunde, d. h. es werden 16 Ziffern benötigt. Die ersten zehn Ziffern entstammen dem Dezimalsystem, 0 bis 9, die folgenden sechs Ziffern werden beginnend mit den ersten Buchstaben des Alphabets bezeichnet, also A bis F.

dual:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hex:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

3.3.1 Umwandlung Hex nach Dezimal

Folgende Hexadezimalzahl ist gegeben: 2AE

$$\begin{array}{rclclcl}
 2 & = & 2 * & 16^2 & = & 512_{(10)} \\
 A & = & 10 * & 16^1 & = & 160_{(10)} \\
 E & = & 14 * & 16^0 & = & 14_{(10)} = 686_{(10)}
 \end{array}$$

3.4 Zeichen und Symbole

\mathbb{C} = komplexe Zahl
 \mathbb{I} = irrationale Zahl
 \mathbb{N} = natürliche Zahl
 \mathbb{Q} = rationale Zahl
 \mathbb{R} = reelle Zahl
 \mathbb{Z} = ganze Zahl

f

= Funktion

x

= Argument, x-Wert, unabhängige Variable

y

= Funktionswert, y-Wert, abhängige Variable

$y = f(x)$

= Funktionsgleichung, Zuordnungsvorschrift

$f(x)$

= spricht man "f von x"

D (oder \mathbb{D})

= Definitionsmenge, Definitionsbereich

W

= Wertemenge, Wertebereich

$f(x) = c$

= konstante Funktion

$f(x) = mx + n$

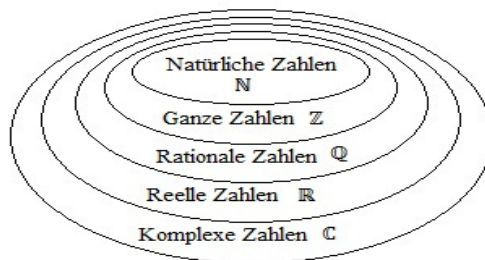
= lineare Funktion

$f(x) = ax^2 + bx + c$

= quadratische Funktion

$f(x) = \frac{a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0}{a_m x^m + b_{m-1} x^{m-1} + \dots + b_1 x + b_0}$

= rationale gebrochene Funktion



Zahlen und Zahlenbereiche

4.1 Natürliche Zahlen

4.1.1 Einführung

Der Begriff Zahl ist vom althochdeutschen Wort "zala" abgeleitet. Dieser Begriff wurde mit "Einschnitt ins Kerbholz" übersetzt. Diese Übersetzung zeigt, welche Bedeutung der Zahlbegriff historisch hatte. Er sollte helfen, die Welt messbar und zählbar zu machen. Die Völker mit einer schriftlichen Kultur haben im Laufe der Geschichte verschiedene Zahlen- und Notationsysteme entwickelt.

Unser heutiges Zahlensystem wird arabisch-indisches System genannt. Es basiert auf dem Dezimalsystem und enthält die Ziffern 0 bis 9. Aus diesen Symbolen oder Zeichen lassen sich nach einfachen Gesetzmässigkeiten beliebige Zahlen bilden. Die beiden Begriffe Kardinalzahlen und Ordinalzahlen beschreiben jeweils die Tätigkeit des Ordnen und des Zählens. Unter Kardinalzahlen versteht man die ganzen Zahlen, mit denen gezählt wird und Mengen beschrieben werden. Beispielsweise spricht man von 2 Hunden, 4 Katzen, 32 Kilometer usw. Die Ordinalzahlen hingegen beschreiben Ordnungen, Rang- und Reihenfolgen. Jemand gewinnt den 2. Preis oder schaut zum 10ten Mal 425 Folge von BibBangTheory.

Natürliche Zahlen N sind **positive** Zahlen, die durch 0 bis 9 symbolisiert dargestellt werden. Natürliche Zahlen können gepaart werden, indem man an den Zahlen 1 bis 9 weitere natürliche Zahlen anfügt (zum Bsp.: 12, 34, 22). Bei der Aufstellung der natürlichen Zahlen ist wie in der Mathematik üblich eine einheitliche Form einzuhalten. So kann/darf man bei einer Definition nicht einfach: 12, 1 33 schreiben !!!

Es ist zwar keine feste Regel dafür manifestiert, aber der Übersichtlichkeit ist eine gewisse Disziplin der Ordnung nicht falsch.

Die Menge der natürlichen **positiven** Zahlen werden wie folgt definiert:

$$N = \{0; 1; 2; 3; \dots\}$$

Natürliche Zahlen sind nur innerhalb eines mehr oder weniger großen Bereichs abzählbar, da sie unendlich sind. Jede natürliche Zahl n hat immer einen unmittelbaren Nachfolger $n + 1$ hat.

4.2 ganze Zahlen

4.2.1 Einführung

Das Ergebnis einer Addition oder einer Multiplikation zweier natürlicher Zahlen ist immer eine weitere natürliche Zahl. Man sagt, dass die Menge der natürlichen Zahlen bezüglich der Addition und der Multiplikation in sich abgeschlossen sind. Achtung: Dies gilt nicht für die Subtraktion und der Division !

Die Subtraktion zweier natürlichen Zahlen ist nur mit den im natürlichen Wertebereich der Zahlen 0

bis 9 möglich **und** wenn der Minuend größer ist als der Subtrahend. Ist jedoch der Minuend größer als der Subtrahend, ist das Ergebnis der Subtraktion keine natürliche Zahl mehr. Deshalb werden die natürlichen Zahlen um die negativen Zahlen ergänzt.
Ganze Zahlen Z erweitern die natürlichen Zahlen N .

4.3 rationale Zahlen

4.3.1 Einführung

Die Menge der natürlichen Zahlen ist abgeschlossen bezüglich der Addition und Multiplikation, nicht jedoch bezüglich der Subtraktion und Division. Deshalb wurden die natürlichen Zahlen zu den ganzen Zahlen erweitert. Die Menge der ganzen Zahlen ist nun auch abgeschlossen. Allerdings ist die Division in den ganzen Zahlen nicht uneingeschränkt möglich.

Deshalb wird der Zahlenbereich nun ein zweites Mal erweitert, so dass er auch abgeschlossen ist. Die neu einzuführenden Zahlen sind die Brüche, die als:

$\frac{a}{b}$
geschrieben werden.

4.4 gebrochene Zahlen

4.4.1 Einführung

Gebrochene Zahlen $Q+$ oder Q^* sind eine Erweiterung der natürlichen Zahlen N , die gestattet, uneingeschränkt zu dividieren. Dazu werden natürliche Zahlen um den **negativen** Zahlenbereich von N sowie um Brüche (rationale Zahlen Q) ergänzt.

Addition und Multiplikation können als "abgeschlossene Operationen" betrachtet werden. Die Summe zweier natürlicher Zahlen ergibt immer eine weitere natürliche Zahl:

$$3 + 5 = 8 \quad | \quad 3 \in \mathbb{N}; 5 \in \mathbb{N}; 8 \in \mathbb{N}$$

$$3 * 5 = 15$$

Minus und Division gelten als "nicht abgeschlossene Operationen". Die Differenz zweier natürlicher Zahlen muss nicht immer eine natürliche ergeben:

$$3 - 5 = -2 \quad | \quad 3 \in \mathbb{N}; 5 \in \mathbb{N} \quad -2 \notin \mathbb{N}$$

$$3 : 5 = 0,6 \quad | \quad 3 \in \mathbb{N}; 5 \in \mathbb{N} \quad 0,6 \notin \mathbb{N}$$

Bonus:

Im Internet habe ich ein etwas verunglücktes Beispiel zu unendliche N gefunden, das ich hier vorstellen, aber auch Kommentieren will.

- man denke sich ein kosmisches Hotel mit unendlich vielen Zimmern vor.
- das Hotel ist *voll* belegt.
- Nun kommt noch ein Gast.

Frage: Kann er in einen voll belegten Hotel noch untergebracht werden? Antwort: = JA!

- da es unendlich viele Zimmer gibt, rückt jeder Gast nur ein Zimmer weiter und das erste wird frei.
- nach dem selben Prinzip können natürlich auch weitere 10 Gäste untergebracht werden.

Kommentar von mir dazu:

Der Sichtwinkel ist hierbei wichtig! Logisch ist es, wenn man von einem *vollen* Hotel spricht, das alle Betten belegt sind. Da aber der Begriff "unendlich", kein Ende oder *voll* definiert ist, können auch "unendlich" viele Betten/Zimmer bezogen werden.

Anders ausgedrückt kann die Zahl unendlich, kann an die Zahl unendlich angeknüpft werden, um wieder eine Menge von unendlich und nicht abzählbaren Zahlen zu bekommen.

4.5 reelle Zahlen

4.5.1 Einführung

Die Menge der rationalen Zahlen ist abgeschlossen bezüglich der Addition, Subtraktion, Multiplikation und Division. Es gibt jedoch Operationen, die aus den rationalen Zahlen herausführen. Eine dieser Operationen ist das Radizieren (wurzelziehen). Die Wurzeln der meisten natürlichen Zahlen sind keine rationalen Zahlen. Dies werde ich später für $\sqrt{2}$ zeigen.

Es existieren also Zahlen, die keine rationalen Zahlen sind. Um diese Lücke zu schließen, benötigt man eine Erweiterung des Zahlenbereichs. Die Zahlen, die diese Lücke auf der Zahlengerade beschreiben, nennt man **irrationale Zahlen**. Sie lassen sich durch unendliche, nichtperiodische Dezimalbrüche darstellen. Die rationalen und irrationalen \mathbb{I} Zahlen ergeben die Menge der reellen Zahlen. Sie wird mit \mathbb{R} bezeichnet. Die positiven reellen Zahlen werden entsprechend mit \mathbb{R}^+ und die negativen mit \mathbb{R}^- bezeichnet.

Bei den irrationalen Zahlen unterscheidet man zwischen algebraischen und transzendenten Zahlen. Eine Zahl heißt algebraisch, wenn die Lösung einer Gleichung der Form:

$$a_x^n + \dots + a_2x^2 + a_1x + a_0 = 0$$

ist.

Zahlen, die nicht algebraisch sind, nennt man transzendent. Transzendente Zahlen sind zum Beispiel die Kreiszahl $\pi = 3,1415\dots$ und die eulersche Zahl $e = 2,71\dots$. Die eulersche Zahl spielt bei den Exponential- und Logarithmusfunktionen eine wichtige Rolle.

4.5.2 Die Quadratwurzel aus 2 ist keine rationale Zahl

Ich will nun zeigen, dass $\sqrt{2}$ keine rationale Zahl ist. Angenommen, dies wäre der Fall, dann ließe sich $\sqrt{2}$ als Bruch: $\sqrt{2} = \frac{a}{b}$ schreiben.

Man kann dabei ohne allgemeine Beschränkung annehmen, dass der Bruch eine Grunddarstellung ist. Die Zahlen a und b sind also teilerfremd. Quadriert man die obige Gleichung, dann erhält man: $2 = \frac{a^2}{b^2}$.

Diese Gleichung kann man mit b^2 multiplizieren: $2 * b^2 = a^2$.

Hieraus folgt, dass a^2 den Teiler 2 hat. Denn a^2 und b^2 sind ganze Zahlen. Somit besitzt auch a den Teiler 2. Man kann deshalb die Zahl a durch $a = 2*k$ ersetzen, wobei k eine ganze Zahl ist.

Also muss auch b^2 den Teiler 2 haben. Dies steht aber im Widerspruch zu der Voraussetzung, dass a und b teilerfremde Zahlen sind. Damit ist die Annahme, $\sqrt{2}$ sei eine rationale Zahl, falsch.

Auf die gleiche Art kann man auch zeigen, dass die Wurzel einer beliebigen Primzahl eine irrationale Zahl ist.

4.6 komplexe Zahlen

4.6.1 Einführung

Komplexe Zahlen \mathbb{C} erweitern den Zahlenbereich des reellen Zahlenbereichs.

Beispiele solcher Zahlen sind: $i, 7 + 3i, 3 - 4i$

Mit fortschreitender Erlangung von neuen Kenntnissen (z. Bsp. auch geprägt von der Nutzung elektronischer Einheiten; womit ich die Einführung des mathematische Binärsystems - n^2 andeuten will), wurde man dadurch motiviert, komplexe Zahlen einzuführen.

Man erkannte, dass Gleichungen wie $x^2 = -1$ nicht lösbar sind. Da es nun aber auch die Zahl -1 (gesprochen: minus eins) in der Zahlentheorie gibt, wurde eine **imaginäre Einheit** eingeführt, die mit **i** - als Definition: $i^2 = -1$ manifestiert wurde. Mit komplexen Zahlen wurde somit das Problem behoben.

4.7 Beispiel aus de.sci.mathematik

Gegeben ist folgende Gleichung:

$$\sqrt[n]{e^{i(\phi)}} = e^{i * (\frac{\phi}{n} + k * 2 * \frac{\pi}{n})} \text{ mit } k = 0, 1, 2, \dots, n - 1.$$

Hier die einfache Herleitung:

Für $n = 2$ und $k = 0$ aus der Gleichung:

e entspricht der Eulerzahl 1

i entspricht der imaginären Zahl: $i^2 = -1$

dann ergibt sich aus e und i : $i^2 = -1$

ϕ entspricht $1 * \text{phi}$

$\frac{\pi}{2}$ entspricht die Hälfte der Kreiszahl π : $\frac{3.14}{2} = 1.57$

1. $\sqrt[2]{1 - 1 * \phi} = 1 - 1 * (\frac{\phi}{2} + 0 * 2 * \frac{\pi}{2})$
2. $\sqrt[2]{1 - 1 * \phi} = 1 - 1 * (\frac{\phi}{2} + 0 * 2 * 1.57)$
3. $\sqrt[2]{1 - \phi} = \frac{1 * 2}{1} * \frac{\phi}{2} \quad | \quad 2 \text{ und } 2 \text{ kürzt sich weg. } (\phi = 1)$
4. $\sqrt[2]{1 - 1} = 1 * 1$
5. $\sqrt[2]{1} = 1 = \sqrt[2]{1} = \sqrt[2]{1}$
6. $1 = 1$

Gegeben ist:

$$1. -1 = (e^{2 * i * \pi})^{\frac{1}{2}} = (1 * e^{2 * i * \pi})^{\frac{1}{2}} = (e^{2 * i * \pi} * e^{2 * i * \pi})^{\frac{1}{2}} = (e^{4 * i * \pi})^{\frac{1}{2}} = 1$$

$$2. -1 = (e^{2*i*\pi})^{\frac{1}{2}} = (e^{4*i*\pi})^{\frac{1}{2}}$$

$$3. -1 = (1^{2*-1*\pi})^{\frac{1}{2}} = (1^{4*-1*\pi})^{\frac{1}{2}}$$

$$4. -1 = (1^{2*-\pi})^{\frac{1}{2}} = (1^{4*-\pi})^{\frac{1}{2}}$$

$$5. -1 = (-3.14)^{\frac{1}{2}} = (-3.14)^{\frac{1}{2}}$$

$$6. -1 = -1.57 = -1.57$$

$$7. -1 = ((-1.57 \implies -1.57) = \text{wahr} = 1) = 1$$

$$8. -1 = 1 = 1$$

nun wird jedes Glied mit -1 multipliziert:

$$9. -1 * -1 = 1 \left\{ \begin{array}{l} 1 * -1 = -1 \\ 1 * -1 = -1 \\ \underbrace{\hspace{1cm}}_{= 0, \text{ kürzt sich weg}} \end{array} \right\} = 1 * -1 = -1$$

Ergebnis: $-1 \mapsto 1 = \text{richtig} !$

Mengen

Mengen können auf zwei verschiedene weisen dargestellt werden:

1. aufzählende Schreibweise:

Alle Elemente einer Menge in geschweifter Klammer: $M = \{1; 2; 3\}$

Wenn in einer Menge ein längeres Intervall existiert, kann man sich durch Schreibweise ... bedienen. Diese Schreibweise kennzeichnet Elemente, die in der Menge *M* vorkommen können, jedoch aus Platzgründen nicht mit aufgeschrieben werden - man könnte es auch als Platzhalter verstehen.

Und hier noch die Schreibweise: $M = \{1; 2; 3; \dots; 10; 11\}$

2. die beschreibende Schreibweise:

Mit dieser Schreibweise wird versucht, Elemente einer Menge mit mathematischen Aussagen zu beschreiben. Erfüllt ein Element eine Aussage, so ist dieses Element der Menge:

$M = \{p \mid "p \text{ ist eine Primzahl}" \}$

Sei Menge M eine mathematisch beschreibende Aussage:

$M = \{z \in N\}$

dann spricht man von einer Menge M, in der "z Element von N ist".

Wenn gilt: $M = \{z \leq 17\}$

dann spricht man von einer Menge, in der nur das Element z kleiner gleich 17 enthalten sein darf/ (oder alle Elemente von z kleiner gleich 17 sind).

Mengendiagramme sind Diagramme, die Elemente in einer geschlossener Umgebung enthalten.

In der realen Welt begegnen uns häufig Abhängigkeiten zwischen zwei Größen.

Als Beispiel hierfür sei die Fläche einer geometrischen Grundkörpers ist abhängig von der Seitenlänge eines Quadrats, oder bei einen Kreis, dessen Radius.

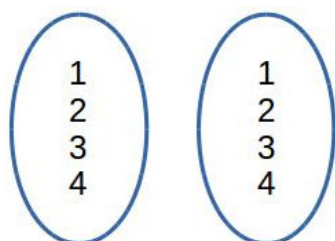
Um die Abhängigkeiten besser verstehen zu können, spricht man auch von einer Zuordnung eines Wertes zu einen anderen Wert.

Beispiel:

Der Hefeteig eines Kuchens hat in der ersten Stunde das doppelte Volumen. In der zweiten Stunde hat der Teig das doppelte Volumen des Vorgängers.

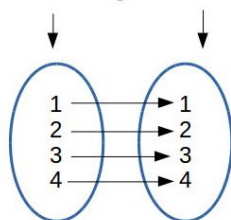
Erst wenn wir verstanden haben, was eine Zuordnung ist, können wir uns mit Funktionen näher beschäftigen. Grund dafür ist, dass eine Funktion nichts anderes als eine Zuordnung mit bestimmten Eigenschaften ist.

Außerdem müssen wir unseren mathematischen Wortschatz um einige Vokabeln erweitern.



Die linke Mengen wird als Definitions(menge) bezeichnet, während die rechte Menge als Werte(menge) bezeichnet wird.

Definitionsmenge Wertemenge

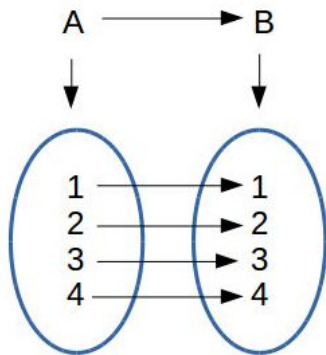


Wie wir bereits wissen, besteht zwischen den beiden Mengen eine Beziehung. Diese Beziehung lässt sich mit Zuordnungspfeilen verdeutlichen.

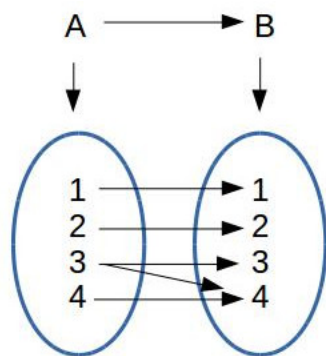
$1 \mapsto 1$

...

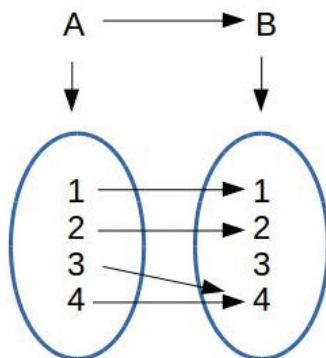
$4 \mapsto 4$



Bei $f : A \mapsto B$ handelt es sich um eine Funktion, da jedem Element x der Menge A genau ein Element y der Menge B zugeordnet ist.



Bei $f : A \mapsto B$ handelt es sich um keine Funktion, da dem Element 3 der Menge A zwei Elemente (3 und 4) der Menge B zugeordnet sind. 4



Bei $f : A \mapsto B$ handelt es sich um eine Funktion, da jedem Element x der Menge A genau ein Element y der Menge B zugeordnet ist.

Dass sich einem Element aus der Menge B zwei Elemente der Menge A zuordnen lassen, spielt keine Rolle. Es handelt sich laut Definition trotzdem um eine Funktion.

Die Erkenntnisse aus den obigen Beispielen lassen sich folgendermaßen zusammenfassen: Eine Funktion liegt vor, wenn von jedem Element x der linken Menge (Definitionsmenge) genau ein Pfeil abgeht. Von wie vielen Pfeilen ein Element y der rechten Menge (Wertemenge) getroffen wird, spielt dagegen für die Definition einer Funktion keine Rolle.

Bezeichnungen und Schreibweisen

Leider verwenden nicht alle Autoren/Lehrer dieselben Begriffe. Es ist deshalb notwendig, dass man

die alternativen Bezeichnungen im Hinterkopf behält, um Verwirrungen beim Lesen verschiedener Mathematiktexte zu vermeiden.

Zwei Funktionen sind genau dann identisch, wenn sie in folgenden Teilen übereinstimmen:

- * Funktionsgleichung
- * Definitionsmenge
- * Wertemenge

Demzufolge sind zwei Funktionen mit gleicher Funktionsgleichung, aber verschiedenen Definitionsmengen oder verschiedenen Wertemengen, nicht identisch und können somit unterschiedliche Eigenschaften besitzen.

Beispiele einer Funktion:

$$y = 2x, D = \{1, 2, 3\}, W = \{2, 4, 8\}$$

Neuronale Netze

Tauchen Sie ein, in die wundersame Welt der künstlichen Intelligenz.

Ich beschreibe hier einen neuronalen Netzwerk Simulator, der auch von nicht-technischen Leuten, einfach zu verstehen ist. Basierend auf backpropagation's Lernen für das hier vorgestellte Tool, ist es möglich, Ihren Computer zu trainieren und zu lernen, was Sie von ihm erwarten.

Ich möchte Ihnen zugleich einen Vorrausblick geben, was uns in naher Zukunft erwartet.

7.1 Arten von Netzwerken

Folgende neurale Netzwerke, werde ich hier noch vorstellen:

- * ein Netzwerk, um die Zahlen 1, 2, und 3 zu erkennen.
- * ein Netzwerk, um die logische Funktion AND zu verarbeiten.
- * ein Netzwerk, um die logische Funktion XOR zu verarbeiten.

7.2 Was sind Neuronale Netzwerke ?

Expert Definition:

Ein neurales Netzwerk ist eine parallel-laufende Informationsstruktur.

7.2.1 Künstliches Neuron

Ein künstliches Neuron bildet die Basis für das Modell der künstlichen neuronalen Netze, ein Modell aus der Neuroinformatik, das durch biologische neuronale Netze motiviert ist. Als konnektionistisches Modell bilden sie in einem Netzwerk aus künstlichen Neuronen ein künstliches neuronales Netz und können so beliebig komplexe Funktionen approximieren, Aufgaben erlernen und Probleme lösen, bei denen eine explizite Modellierung schwierig bis nicht durchzuführen ist. Beispiele sind die Gesichts- und Spracherkennung.

Als Modell aus dem biologischen Vorbild der Nervenzelle entstanden, kann es mehrere Eingaben verarbeiten und entsprechend über seine Aktivierung reagieren. Dazu werden die Eingaben gewichtet an eine Ausgabefunktion übergeben, welche die Neuronenaktivierung berechnet. Ihr Verhalten wird ihnen im Allgemeinen durch Einlernen unter Verwendung eines Lernverfahrens gegeben.

7.2.2 Biologische Motivation

Motiviert sind künstliche Neuronen durch die Nervenzellen der Säugetiere, die auf die Aufnahme und Verarbeitung von Signalen spezialisiert sind. Über Synapsen werden Signale elektrisch oder chemisch an andere Nervenzellen oder Effektorzellen (etwa zur Muskelkontraktion) weitergeleitet.

Eine Nervenzelle besteht aus dem Zellkörper, Axon und den Dendriten. Dendriten sind kurze Zellfortsätze, die stark verzweigt für die Aufnahme von Signalen anderer Nervenzellen oder Sinneszellen sorgen. Das Axon funktioniert als Signalausgang der Zelle und kann eine Länge bis 1 m erreichen. Der Übergang der Signale erfolgt an den Synapsen, welche erregend oder hemmend wirken können. Die Dendriten der Nervenzelle leiten die eingehenden elektrischen Erregungen an den Zellkörper weiter. Erreicht die Erregung einen gewissen Grenzwert und übersteigt ihn, entlädt sich die Spannung und pflanzt sich über das Axon fort (Alles-oder-nichts-Gesetz).

Die Verschaltung dieser Nervenzellen bildet die Grundlage für die geistige Leistung des Gehirns. Das Zentralnervensystem des Menschen besteht nach Schätzungen aus 10^{10} bis 10^{12} Nervenzellen, die durchschnittlich 10.000 Verbindungen besitzen – das menschliche Gehirn kann also mehr als 10^{14} Verbindungen besitzen. Das Aktionspotential im Axon kann sich mit einer Geschwindigkeit bis zu 100 m/s fortpflanzen.

Im Vergleich zu Logikgattern zeigt sich auch die Effizienz von Neuronen. Während Gatter im Nanosekunden-Bereich (10^{-9}) schalten, unter einem Energieverbrauch von 10^{-6} Joule (Daten von 1991), reagieren Nervenzellen im Millisekunden-Bereich (10^{-3}) und verbrauchen lediglich eine Energie von 10^{-16} Joule. Trotz der augenscheinlich geringeren Werte in der Verarbeitung durch Nervenzellen können rechnergestützte Systeme nicht an die Fähigkeiten biologischer Systeme heranreichen. Die Vorteile und Eigenschaften von Nervenzellen motivieren das Modell der künstlichen Neuronen.

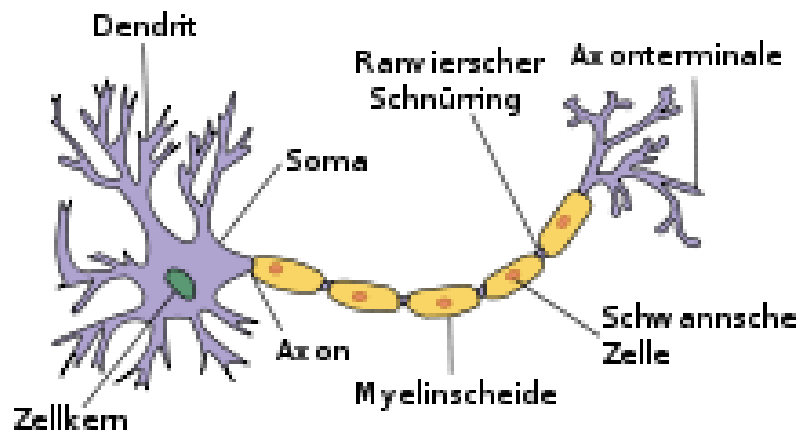


Abbildung 7.1: Schema einer Nervenzelle

Viele Modelle und Algorithmen zu künstlichen neuronalen Netzen entbehren dennoch einer direkt plausiblen, biologischen Motivierung. Dort findet sich diese nur im Grundgedanken der abstrakten Modellierung der Nervenzelle.

7.2.3 Modellierung

Mit der Biologie als Vorbild wird nun durch eine passende Modellbildung eine für die Informatik verwendbare Lösung gefunden. Durch eine grobe Verallgemeinerung wird das System vereinfacht – unter Erhaltung der wesentlichen Eigenschaften.

Die Synapsen der Nervenzelle werden hierbei durch die Addition gewichteter Eingaben abgebildet, die Aktivierung des Zellkerns durch eine Aktivierungsfunktion mit Schwellenwert.

7.2.4 Bestandteile

Ein künstliches Neuron j kann durch vier Basiselemente beschrieben werden:

1. **Wichtung:** Die Gewichte w_{ij} bestimmen den Grad des Einflusses, den die Eingaben des Neurons in der Berechnung der späteren Aktivierung einnehmen. Abhängig von den Vorzeichen der Gewichte kann eine Eingabe hemmend (inhibitorisch) oder erregend (exzitatorisch) wirken. Ein Gewicht von 0 markiert eine nicht existente Verbindung zwischen zwei Knoten.
2. **Übertragungsfunktion:** Die Übertragungsfunktion Σ berechnet anhand der Wichtung der Eingaben die Netzeingabe des Neurons.
3. **Aktivierungsfunktion:** Die Ausgabe des Neurons wird schließlich durch die Aktivierungsfunktion φ bestimmt. Die Aktivierung wird beeinflusst durch die Netzeingabe aus der Übertragungsfunktion sowie einem Schwellenwert.

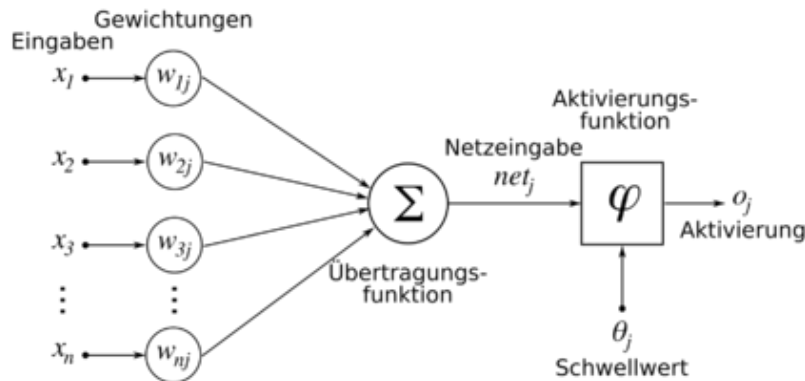


Abbildung 7.2: Darstellung eines künstlichen Neurons mit seinen Elementen

4. **Schwellenwert:** Das Addieren eines Schwellenwerts θ_j zur Netzeingabe verschiebt die gewichteten Eingaben. Die Bezeichnung bestimmt sich aus der Verwendung einer Schwellenwertfunktion als Aktivierungsfunktion, bei der das Neuron aktiviert wird, wenn der Schwellenwert überschritten ist. Die biologische Motivierung dabei ist das Schwellenpotential bei Nervenzellen. Mathematisch gesehen wird die Trennebene, die den Merkmalsraum auftrennt, durch einen Schwellenwert mit einer Translation verschoben.

Durch einen Verbindungsgraphen werden folgende Elemente festgelegt:

1. **Eingaben:** Eingaben x_i können einerseits aus dem beobachteten Prozess resultieren, dessen Werte dem Neuron übergeben werden, oder wiederum aus den Ausgaben anderer Neuronen stammen. Sie werden auch so dargestellt:
2. **Aktivierung oder Ausgabe:** Das Ergebnis der Aktivierungsfunktion wird analog zur Nervenzelle als Aktivierung o_j des künstlichen Neurons j bezeichnet.



Abbildung 7.3: Symbol eines aktivierten Neurons

7.3 Mathematische Definition

Das künstliche Neuron als Modell wird in der Literatur meist auf dem folgenden Weg eingeführt: Zuerst wird die Netzeingabe

net_j des künstlichen Neurons j durch

$$\text{net}_j = \sum_{i=1}^n x_i w_{ij}$$

definiert und damit die Aktivierung

$$o_j \text{ durch} \\ o_j = \varphi(\text{net}_j - \theta_j) .$$

Dabei ist

n die Anzahl der Eingaben und
 x_i die Eingabe i , die sowohl diskret als auch stetig sein kann.

7.3.1 on-Neuron

Alternativ kann der Schwellenwert auch durch Hinzufügen eines weiteren Eingangs x_0 , einem sogenannten on-Neuron oder auch Bias, dargestellt werden. Dieser hat den konstanten Wert $x_0 = 1$. Der Schwellenwert ist dann die Gewichtung dieses Eingangs $w_{0j} = -\theta_j$. Eine spezielle Behandlung des Schwellenwerts kann so entfallen und vereinfacht die Behandlung in den Lernregeln.

Zusätzlich zu den echten Eingängen wird nun der des on-Neurons mit einberechnet:

$$\text{net}_j = \sum_{i=0}^n x_i w_{ij}$$

Bei der Aktivierung o_j kann damit auf eine spezielle Behandlung des Schwellenwerts verzichtet werden:

$$o_j = \varphi(\text{net}_j)$$

7.4 Merkmalsraum

Die Mustererkennung untersucht die automatische Klassifizierung, also wie man Objekte automatisch in Klassen einordnen kann. Um Objekte zu unterscheiden, bestimmt man zunächst eine Reihe

von Merkmalen, in denen sie sich möglichst stark unterscheiden. Dann misst man für jedes zu klassifizierende Objekt diese Merkmale und schreibt die Messergebnisse untereinander in einen Vektor, den sogenannten Merkmalsvektor. Man erhält dadurch für jedes Objekt einen Vektor mit so vielen Einträgen, wie Merkmale betrachtet werden. Jeder dieser Vektoren bezeichnet einen Punkt im Merkmalsraum; der Merkmalsraum hat also so viele Dimensionen wie Merkmale betrachtet werden. Gesucht ist nun eine Funktion, die den Merkmalsraum in mehrere Klassen zerlegt, ein sogenannter Klassifikator.

7.4.1 Merkmalsvektor

Ein Merkmalsvektor fasst die (numerisch) parametrisierbaren Eigenschaften eines Musters in vektorieller Weise zusammen. Verschiedene, für das Muster charakteristische Merkmale bilden die verschiedenen Dimensionen dieses Vektors. Die Gesamtheit der möglichen Merkmalsvektoren nennt man den Merkmalsraum. Merkmalsvektoren erleichtern eine automatische Klassifikation, da sie die zu klassifizierenden Eigenschaften stark reduzieren (statt eines kompletten Bildes muss zum Beispiel nur ein Vektor aus 10 Zahlen betrachtet werden). Häufig dienen sie als Eingabe für eine Clusteranalyse.

In der Mustererkennung gebräuchliche Merkmalsräume sind mehrdimensionale reelle Vektorräume: \mathbb{R}^d . Die Dimension d des Raums entspricht der Anzahl der untersuchten Merkmale und kann sehr groß sein.

7.4.2 Musterklassifikation

In der Musterklassifikation werden Muster anhand von ihren parametrisierbaren Eigenschaften, den Merkmalsvektoren, automatisch klassifiziert. Je besser die Merkmale gewählt wurden und je mehr Trainingsmaterial (also je größer die Stichprobe) vorhanden ist, desto besser gelingt eine Klassifikation. Eine größere Dimension in den Merkmalsvektoren bedeutet dabei einen größeren Bedarf an Trainingsmaterial, also auch einen größeren Trainingsaufwand und eine größere Trainingsdauer. Aber dafür erzielt man auch bessere Klassifikationsraten, also eine bessere Klassifikatorqualität. Eine geringe Anzahl von Dimensionen bedeutet dabei ein schnelleres Training und eine kleinere Stichprobe, aber auch geringere Qualität.

7.4.3 Klassifikator

Ein Klassifikator ist ursprünglich der Bearbeiter eines Sachkatalogs im Bibliothekswesen.[1] Allgemein bezeichnet Klassifikator eine Instanz, die Objekte klassifiziert, d. h. in Kategorien einordnet.

Welcher Art eine solche Instanz ist, ist vom betrachteten Szenario abhängig: In der Informatik ist ein Klassifikator häufig ein Algorithmus oder ein Programmobjekt, speziell in der Mustererkennung eine mathematische Funktion, die einen Merkmalsraum auf eine Menge von Klassen abbildet. Auch mechanische Bauteile, zum Beispiel zum Sortieren von Münzen nach Größe, können Klassifikatoren darstellen.

In vielen Fällen ist die Unterscheidung zwischen Verfahren und Instanz nicht notwendig oder sinnvoll, weshalb der Begriff Klassifikator häufig gleichbedeutend mit Klassifikationsverfahren verwendet wird.

7.4.4 Klassifikationsverfahren

Klassifikationsverfahren auch Klassifizierungsverfahren sind Methoden und Kriterien zur Einteilung (Klassierung) von Objekten oder Situationen in Klassen, das heißt zur Klassifizierung. Ein solches Verfahren wird auch als Klassifikator bezeichnet. Viele Verfahren lassen sich als Algorithmus implementieren; man spricht dabei auch von maschineller oder automatischer Klassifikation. Klassifikationsverfahren sind immer anwendungsbezogen, so dass viele verschiedene Methoden existieren.

Im engen Sinne stehen im Gegensatz zu den Klassifikationsverfahren die Klassierungsverfahren welche dem Einordnen von Objekten in bereits existierende Klassen dienen. Umgangssprachlich wird jedoch zwischen Klassifizieren und Klassieren kein Unterschied gemacht.

Klassifikationsverfahren spielen unter anderem bei der Mustererkennung, in der Künstlichen Intelligenz und der Dokumentationswissenschaft beziehungsweise dem Information Retrieval eine Rolle. Zur Beurteilung eines Klassifikators können verschiedene Kenngrößen ermittelt werden.

Arten von Klassifikationsverfahren

Da eine streng hierarchische Einteilung von Klassifikationsverfahren kaum möglich ist, lassen sie sich am besten anhand verschiedener Eigenschaften einteilen:

- Manuelle und automatische Verfahren
- Numerische und nichtnumerische Verfahren
- Statistische und verteilungsfreie Verfahren
- Überwachte und nichtüberwachte Verfahren
- Fest dimensionierte und lernende Verfahren
- Parametrische und nichtparametrische Verfahren

Manuelle und automatische Verfahren

Bei automatischen Verfahren findet die Klassifizierung mittels eines automatischen Prozesses durch Software statt. Der Prozess der maschinellen Klassifikation kann als formale Methode des Entscheidens in neuen Situationen aufgrund erlernter Strukturen bezeichnet werden. Die maschinelle Klassifikation ist ein Teilgebiet des Maschinellen Lernens.

Genauer ist dies die Erzeugung eines Algorithmus (der lernende Algorithmus), der - angewandt auf bekannte und schon klassifizierte Fälle (die Datenbasis) - Strukturen berechnet. Diese neu erlernten Strukturen ermöglichen es einem weiteren Algorithmus (der auswertende Algorithmus), einen neuen und bisher unbekannten Fall aufgrund der beobachteten Attribute und deren Ausprägungen einer der bekannten Ziel-Klassen zuzuordnen.

Statistische und verteilungsfreie Verfahren

Statistische Verfahren basieren auf Dichteberechnungen und Wahrscheinlichkeiten, während verteilungsfreie Verfahren klare Trennflächen zur Trennung der Klassen benutzen. Die Grenzen zwischen den einzelnen Klassen im Merkmalsraum können durch eine Diskriminanzfunktionen angegeben werden.

Beispiele für statistische Verfahren sind der Bayes-Klassifikator, der Fuzzy-Pattern-Klassifikator oder Kerndichteschätzer. Die Berechnung von Trennflächen ist durch sogenannte Support-Vector-Maschinen möglich.

Überwachte und nicht überwachte Verfahren

Das Erzeugen von Strukturen aus vorhandenen Daten wird auch als Mustererkennung, Diskriminierung oder überwachtes Lernen bezeichnet. Dabei werden Klasseneinteilungen vorgegeben (dies kann auch durch Stichproben geschehen). Im Gegensatz dazu existiert nichtüberwachtes Lernen, bei dem die Klassen der Daten nicht vorgegeben sind, sondern auch diese erlernt werden müssen. Dabei können allerdings beim Bestärkenden Lernen (engl.: reinforcement learning) Informationen hinzukommen, ob eine Klasseneinteilung richtig oder falsch war. Ein Beispiel für unüberwachte Verfahren ist die Clusteranalyse.

Parametrische und nichtparametrische Verfahren

Parametrische Verfahren beruhen auf parametrischen Wahrscheinlichkeitsdichten, während nichtparametrischen Verfahren (z.B. Nächste-Nachbarn-Klassifikation) auf lokalen Dichteberechnungen basieren.

Überwachtes Lernen

Mit Lernen ist dabei die Fähigkeit gemeint, Gesetzmäßigkeiten nachzubilden. Die Ergebnisse sind durch Naturgesetze oder Expertenwissen bekannt und werden benutzt, um das System anzulernen. Ein Lernalgorithmus versucht, eine Hypothese zu finden, die möglichst zielsichere Voraussagen trifft. Unter Hypothese ist dabei eine Abbildung zu verstehen, die jedem Eingabewert den vermuteten Ausgabewert zuordnet. Dazu verändert der Algorithmus die freien Parameter der gewählten Hypothesenklasse. Oft wird als Hypothesenklasse die Menge aller Hypothesen, die durch ein bestimmtes künstliches neuronales Netzwerk modelliert werden kann, verwendet. In diesem Fall sind die frei wählbaren Parameter die Gewichte der Neuronen. Beim überwachten Lernen werden diese Gewichte derart angepasst, dass die Ausgabe der Neuronen denen eines vorgegebenen Teaching Vectors (engl., Lernvektor) möglichst nahekommt. Die Methode richtet sich also nach einer im Vorhinein festgelegten zu lernenden Ausgabe, deren Ergebnisse bekannt sind. Die Ergebnisse des Lernprozesses können mit den bekannten, richtigen Ergebnissen verglichen, also „überwacht“, werden.

Um zu wissen, wann eine Hypothese zielsicher ist, wird ein Fehlermaß eingeführt, das minimiert werden soll. Eine beliebte Wahl ist der mittlere quadratische Fehler aller Trainingsdaten. Ein Lernschritt könnte wie folgt aussehen:

1. Anlegen der Eingabe
2. Verarbeitung der Eingabe (Propagierung)
3. Vergleich der Ausgabe mit dem erwünschten Wert (Fehler)
4. Verkleinern des Fehlers durch Modifikation der Gewichte (z. B. mit Backpropagation)

Nach diesem Training bzw. Lernprozess sollte das System in der Lage sein, zu einer unbekannten, den gelernten Beispielen ähnlichen Eingabe, eine korrekte Ausgabe zu liefern.

Um diese Fähigkeit zu testen, wird das System validiert. Eine Möglichkeit ist, die verfügbaren Daten in ein Trainingsset und ein Testset zu unterteilen. Das Ziel ist es, das Fehlermaß im Testset, mit dem nicht trainiert wird, zu minimieren. Häufig kommen dafür Kreuzvalidierungsverfahren zur Anwendung.

Besitzt das Modell sehr viele Parameter (Gewichte) oder sind nur wenige Trainingsdaten vorhanden, kommt es leicht zur Überanpassung. Das zeigt sich, wenn der Fehler im Trainingsset zwar weiterhin sinkt, aber derjenige im Testset wieder zu steigen beginnt, weil die bekannten Daten einzeln gelernt werden (anstelle der allgemeinen Regel dahinter). Oft wird genau dieser Zeitpunkt abgewartet, um den Trainingsvorgang zu stoppen. Damit wird aber das Testset beim Training verwendet. Zur Beurteilung wird daher ein drittes Validierungsset eingeführt.

8.1 Empirische Daten

Empirie (vom griechischen: *empeiría* ‚Erfahrung, Erfahrungswissen‘) ist eine methodisch-systematische Sammlung von Daten. Auch die Erkenntnisse aus empirischen Daten werden manchmal kurz Em-

pirie genannt.

8.2 mittlerer quadratische Fehler

Die mittlere quadratische Abweichung, auch der mittlere quadratische Fehler genannt und MQF oder MSE (aus dem englischen für mean squared error) abgekürzt, ist ein Begriff der mathematischen Statistik. Er gibt in der Schätztheorie an, wie sehr ein Punktschätzer um den zu schätzenden Wert streut. Damit ist er ein zentrales Qualitätskriterium für Schätzer.

8.3 Punktschätzer

Als Punktschätzer bezeichnet man eine Schätzfunktion, die jeder Stichprobe einen Wert zuordnet, der eine gewisse Eigenschaft des zugrundeliegenden Wahrscheinlichkeitsmaßes schätzen soll. In den meisten Anwendungen ist die interessierende Größe ein Parameter der Wahrscheinlichkeitsverteilung der Beobachtungen (wie z.B. der Mittelwert)

8.4 Schätzfunktion

Eine Schätzfunktion dient in der mathematischen Statistik dazu, aufgrund von vorhandenen empirischen Daten einer Stichprobe einen Schätzwert zu ermitteln und dadurch Informationen über unbekannte Parameter einer Grundgesamtheit zu erhalten. Schätzfunktionen sind die Basis zur Berechnung von Punktschätzungen und zur Bestimmung von Konfidenzintervallen mittels Bereichsschätzern und werden als Teststatistiken in Hypothesentests verwendet. Sie sind spezielle Stichprobenfunktionen und können durch Schätzverfahren, z. B. die Methode der kleinsten Quadrate bestimmt werden.

Im Rahmen der Entscheidungstheorie können Schätzfunktionen auch als Entscheidungsfunktionen bei Entscheidungen unter Unsicherheit betrachtet werden.

8.5 Grundgesamtheit

Bezeichnet zum Beispiel die Gesamtheit von Populationen. Die Menge aller Einheiten (auch Merkmalsträger, Untersuchungseinheit) mit übereinstimmenden Identifikationskriterien (sachlich, räumlich und zeitgleich).

Die Einheit ist Träger der Informationen für die Untersuchung

8.6 Stichprobe

Als Stichprobe bezeichnet man eine Teilmenge einer Grundgesamtheit, die unter bestimmten Gesichtspunkten ausgewählt wurde.

8.6.1 Auswahlverfahren

Ein Auswahlverfahren ist die Art und Weise, wie die Elemente der Stichprobe möglichst zweckmäßig ausgewählt werden. Es gibt verschiedene Auswahlverfahren, die nachfolgend beschrieben werden.

Zufallsauswahl

Eine Zufallsstichprobe ist notwendig, wenn die Stichprobe repräsentativ sein soll, d. h. wenn von ihr nach dem Induktionsprinzip auf die Grundgesamtheit geschlossen werden soll, da es oft nicht möglich ist, die Grundgesamtheit (etwa die Gesamtbevölkerung oder alle Exemplare eines bestimmten Produkts) zu untersuchen.

Bewusste Auswahl

Bei einer systematischen Stichprobenziehung werden bereits bekannte Informationen über die auszuwählenden Fälle genutzt. Die Auswahl erfolgt anhand von Listen und festgelegten Regeln. Mathematisch-statistische Modelle, etwa die Berechnung der Einschlusswahrscheinlichkeit, sind bei bewussten Auswahlen nicht anwendbar. Systematische Auswahlverfahren kommen zum Beispiel im kommerziellen Bereich vor, wenn es auf Repräsentativität nicht ankommt.

Willkürliche Auswahl

Bei willkürlichen Stichproben werden Elemente aus der Grundgesamtheit (etwa von einem Interviewer) mehr oder weniger willkürlich in die Stichprobe aufgenommen. Die Auswahl liegt im Ermessen des Interviewers.

8.7 Systematische Stichprobe

Als Systematische Stichprobe (auch Bewusste Auswahl) bezeichnet man Auswahlverfahren, bei denen subjektive Erwägungen die Auswahl der Zielpersonen bestimmen.

Beispiel: Alle Mitarbeiter mit mehr als 10 Jahren Betriebszugehörigkeit.

Es werden Vorinformationen über die auszuwählenden Fälle genutzt. Verallgemeinerungen sind auf der Basis mathematisch-statistischer Modelle bei bewussten Auswahlen nicht möglich.

8.8 Stichprobenvariable

An dieser Stelle setzt die statistische Modellierung an. Die Stichprobenvariable X_i , eine Zufallsvariable, beschreibt mit ihrer Verteilung die Wahrscheinlichkeit, mit der eine bestimmte Merkmalsausprägung bei der i Ziehung aus der Grundgesamtheit auftritt. Jeder Beobachtungswert x_i ist die Realisierung einer Stichprobenvariable X_i .

8.9 Stichprobenfunktion

Die Definition von Stichprobenvariablen X_i erlaubt die Definition von Stichprobenfunktionen analog z. B. zu Kennwerten aus der deskriptiven Statistik:

8.9.1 Arithmetisches Mittel - Formel

$$\bar{x} = \frac{x_1 + \dots + x_n}{n}$$

8.9.2 Stichprobenfunktion - Formel

$$\bar{X} = \frac{X_1 + \dots + X_n}{n}$$

8.10 Stichprobenverteilung

Unter Stichprobenverteilung versteht man die Verteilung einer Stichprobenfunktion $g(X_1, \dots, X_n)$ über alle möglichen Stichproben aus der Grundgesamtheit. Die Stichprobenfunktion g ist in der Regel eine Schätzfunktion für einen unbekannten Parameter der Grundgesamtheit oder eine Teststatistik für eine Hypothese über einen unbekannten Parameter der Grundgesamtheit. Daher spricht man statt von Stichprobenverteilung auch einfach von der Verteilung einer Schätzfunktion oder Teststatistik. Die Verteilung der Stichprobenfunktion dient der Gewinnung von Aussagen über unbekannte Parameter in der Grundgesamtheit aufgrund einer Stichprobe.

8.11 Hypothese

In der Statistik bezeichnet man mit Hypothese eine Annahme, die mit Methoden der mathematischen Statistik auf Basis empirischer Daten geprüft wird. Man unterscheidet als Gegensatzpaar Nullhypothese und Alternativhypothese (auch Gegenhypothese). Häufig sagt die Nullhypothese aus, dass kein Effekt bzw. Unterschied vorliegt oder dass ein bestimmter Zusammenhang nicht besteht. Diese These soll verworfen werden, so dass die Alternativhypothese als Möglichkeit übrig bleibt. Durch dieses indirekte Vorgehen soll die Wahrscheinlichkeit für eine irrtümliche Verwerfung der Nullhypothese kontrolliert klein bleiben. Oft entsteht jedoch Verwirrung beim Anwender, weil dieses Vorgehen die Möglichkeit nahelegt, dass – sofern die Nullhypothese nicht verworfen und die Alternativhypothese damit nicht angenommen werden kann – die Nullhypothese als erwiesen gilt. Dies ist allerdings nicht der Fall.

8.11.1 Alternativhypothese

Als Alternativhypothese H_1 oder H_A bezeichnet man in der empirischen Wissenschaft häufig eine durch Beobachtungen oder Überlegungen begründete Annahme oder Vermutung, die zur Erklärung bestimmter Phänomene dient und die einer möglicherweise verbreiteten Annahme oder Vermutung (nämlich der Nullhypothese) entgegensteht. Insofern kann die Alternativhypothese als innovativ betrachtet werden.

Formal zerlegen die Null- und Alternativhypothese einen Parameterraum Θ in zwei disjunkte nicht

leere Teilmengen Θ_0 und Θ_1 . Die Nullhypothese beinhaltet die Aussage, dass der unbekannte Parameter θ aus Θ_0 stammt, und die Alternativhypothese, dass der unbekannte Parameter θ aus Θ_1 stammt.

$H_0: \theta \in \Theta_0$ vs.

$H_1: \theta \in \Theta_1$

8.12 Teststatistik

Als Teststatistik (synonyme Begriffe: Testgröße, Prüfgröße, Prüffunktion) bezeichnet man eine bestimmte Stichprobenfunktion, die bei einem Hypothesentest dazu verwendet wird, die Testentscheidung - also Ablehnen oder Nichtablehnen der Nullhypothese - zu treffen.

Als Prüfwert wird die Realisation einer Teststatistik anhand einer Stichprobe bezeichnet.

8.12.1 Statistische Signifikanz

Statistisch signifikant wird das Ergebnis eines statistischen Tests genannt, wenn Stichprobendaten so stark von einer vorher festgelegten Annahme (der Nullhypothese) abweichen, dass diese Annahme nach einer vorher festgelegten Regel verworfen wird.

8.12.2 Verwendung bei festem Signifikanzniveau

Vor der Durchführung des Tests, das heißt auch vor der Ziehung der hierzu benötigten Stichprobe, ist die Teststatistik T eine Zufallsvariable, deren Wahrscheinlichkeitsverteilung von jener der Stichprobenvariablen X_1, X_2, \dots, X_n abhängt, wobei n der Stichprobenumfang ist. Unter der Annahme, dass die Nullhypothese (H_0) richtig ist, wird für die Verteilung der Teststatistik je nach Testverfahren ein bestimmtes Verteilungsmodell angenommen, dessen Verteilungsparameter sich aus der Nullhypothese ergeben. Anhand dieser angenommenen Verteilung sowie des zuvor festgelegten Signifikanzniveaus wird zugleich der Ablehnbereich bestimmt. Nun wird die Stichprobe gezogen und aus den sich dabei ergebenden Stichprobenwerten der konkrete Wert t der Teststatistik errechnet.

Zur Ablehnung der Nullhypothese kommt es genau dann, wenn t in den Ablehnbereich fällt; anderenfalls wird unter dem verwendeten Signifikanzniveau die Nullhypothese beibehalten. Wenn nämlich die Nullhypothese gilt und damit die unterstellte Verteilung der Teststatistik als richtig angenommen werden kann, entspricht die Wahrscheinlichkeit, dass die Testgröße in den Ablehnbereich fällt und somit die Nullhypothese fälschlich abgelehnt wird (sogenannter Fehler 1. Art), genau dem festgelegten Signifikanzniveau. Das Fallen der Testgröße in den Ablehnbereich ist gleichbedeutend mit der (je nach Testproblem) Über- bzw. Unterschreitung eines bestimmten Schwellenwertes, der auch als „Kritischer Wert“ bezeichnet wird.

8.12.3 Fehler 1. Art

Der Fehler 1. Art oder α -Fehler (Alpha-Fehler) ist ein Fachbegriff der Statistik. Er bezieht sich auf eine Methode der mathematischen Statistik, den sogenannten Hypothesentest. Beim Test einer Hypothese liegt ein Fehler 1. Art vor, wenn die Nullhypothese zurückgewiesen wird, obwohl sie in Wirklichkeit wahr ist (beruhend auf falsch positiven Ergebnissen).

Die Ausgangshypothese H_0 (Nullhypothese) ist hierbei die Annahme, die Testsituation befinde

sich im „Normalzustand“. Wird also dieser Normalzustand nicht erkannt, obwohl er tatsächlich vorliegt, ergibt sich ein Fehler 1. Art. Beispiele für einen Fehler 1. Art sind:

- der Patient wird als krank angesehen, obwohl er in Wirklichkeit gesund ist (Nullhypothese: der Patient ist gesund),
- der Angeklagte wird als schuldig verurteilt, obwohl er in Wirklichkeit unschuldig ist (Nullhypothese: der Angeklagte ist unschuldig),
- der Person wird kein Zugang gewährt, obwohl sie eine Zugangsberechtigung hat (Nullhypothese: die Person hat Zugangsberechtigung)

Die vor einem Test bzw. einer Untersuchung festgelegte maximale Wahrscheinlichkeit, bei einer auf dem Ergebnis des Tests fußenden Entscheidung einen solchen Fehler 1. Art zu begehen (Risiko 1. Art), nennt man auch Signifikanzniveau oder Irrtumswahrscheinlichkeit. In der Regel wählt man ein Signifikanzniveau von 5 % (signifikant) oder 1 % (sehr signifikant).

Die andere mögliche Fehlentscheidung, nämlich die Alternativhypothese H_1 zurückzuweisen, obwohl sie wahr ist, heißt Fehler 2. Art.

	Wahrer Sachverhalt: H_0 (Es gibt keinen Unterschied)	Wahrer Sachverhalt: H_1 (Es gibt einen Unterschied)
durch einen statistischen Test fällt eine Entscheidung für: H_0	richtige Entscheidung (Spezifität) Wahrscheinlichkeit: $1-\alpha$	Fehler 2. Art Wahrscheinlichkeit: β
durch einen statistischen Test fällt eine Entscheidung für: H_1	Fehler 1. Art Wahrscheinlichkeit: α	richtige Entscheidung (Sensitivität) Wahrscheinlichkeit: $1-\beta$

Aktivierungsfunktionen

Als Aktivierungsfunktion φ können verschiedene Funktionstypen verwendet werden, abhängig von der verwendeten Netztopologie. Eine solche Funktion kann nicht-linear, zum Beispiel sigmoid, stückweise linear oder eine Sprungfunktion sein. Im Allgemeinen sind Aktivierungsfunktionen monoton steigend.

Lineare Aktivierungsfunktionen unterliegen einer starken Beschränkung, da eine Komposition linearer Funktionen durch arithmetische Umformungen durch eine einzige lineare Funktion dargestellt werden kann. Für mehrschichtige Verbindungsnetzwerke sind sie deswegen nicht geeignet und finden so nur in einfachen Modellen Anwendung.

Beispiele für grundlegende Aktivierungsfunktionen sind:

9.1 Schwellwertfunktion

Die Schwellenwertfunktion (engl. hard limit), wie sie im Folgenden definiert ist, nimmt nur die Werte 0 und 1 an. Den Wert 1 für die Eingabe $v \geq 0$, sonst 0. Bei subtraktiver Verwendung eines Schwellenwerts θ wird die Funktion nur aktiviert, wenn die zusätzliche Eingabe den Schwellenwert übersteigt. Ein Neuron mit einer solchen Funktion wird auch McCulloch-Pitts-Zelle genannt. Sie spiegelt die Alles-oder-nichts-Eigenschaft des Modells wider.

$$\varphi^{\text{hlim}}(v) = \begin{cases} 1 & \text{wenn } v \geq 0 \\ 0 & \text{wenn } v < 0 \end{cases}$$

9.2 Stückweise lineare Funktion

Die hier verwendete stückweise lineare Funktion (engl. piecewise linear) bildet ein begrenztes Intervall linear ab, die äußeren Intervalle werden auf einen konstanten Wert abgebildet:

$$\varphi^{\text{pwl}}(v) = \begin{cases} 1 & \text{wenn } v \geq \frac{1}{2} \\ v + \frac{1}{2} & \text{wenn } -\frac{1}{2} < v < \frac{1}{2} \\ 0 & \text{wenn } v \leq -\frac{1}{2} \end{cases}$$

9.3 Sigmoid Funktion

Eine Sigmoidfunktion, Schwanenhalsfunktion oder S-Funktion ist eine mathematische Funktion mit einem S-förmigen Graphen. Oft wird der Begriff Sigmoidfunktion auf den Spezialfall logistische Funktion bezogen, die durch die Gleichung:

$$\text{sig}(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t} = \frac{1}{2} * (1 + \tanh \frac{t}{2})$$

$$\varphi_a^{\text{sig}}(v) = \frac{1}{1 + \exp(-av)}.$$

Die Werte der obigen Funktionen liegen im Intervall $[0, 1]$. Für das Intervall $[-1, +1]$ lassen sich diese Funktionen entsprechend definieren.

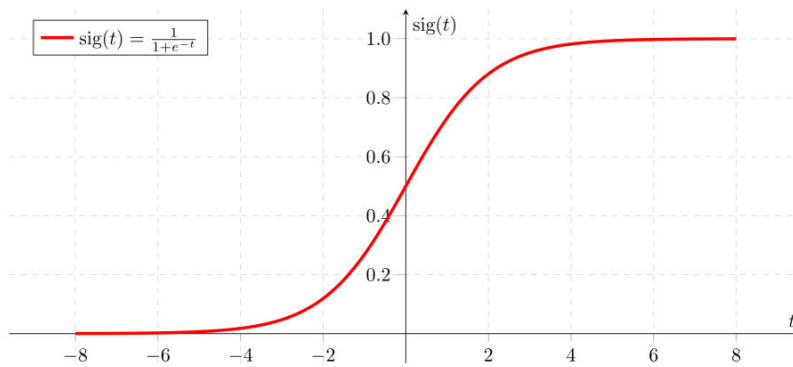


Abbildung 9.1: Graph der Sigmoid-Funktion

9.4 Darstellung boolescher Funktionen

Mit künstlichen Neuronen lassen sich boolesche Funktionen darstellen. So können die drei Funktionen Konjunktion (and), Disjunktion (or) und Negation (not) unter Verwendung einer Schwellenfunktion φ^{hlim} wie folgt repräsentiert werden:

Konjunktion	Disjunktion	Negation
<p>Neuron, das die Konjunktion repräsentiert</p>	<p>Neuron, das die Disjunktion repräsentiert</p>	<p>Neuron, das die Negation repräsentiert</p>

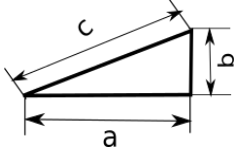
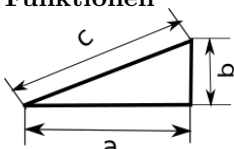
Formelsammlung

Anbei von mir für nützliche empfundene Formeln und Schaubilder:

10.1 Umrechnungen

Arbeit	$1J = 1 = \frac{W}{s} = 1Nm$	-
Leistung	$1W = 1 * \frac{Nm}{s}$	$1PS = 735,49875W$
Wärme	$1J = 1 * \frac{W}{s}$	$1kcal = 4,187kJ$ Wasser: $c = 4,187 \frac{kJ}{kg * K}$
Kraft, Druck	$1N = \frac{W * s}{m}$; $1bar = 10^5 * \frac{N}{m^2}$	$1kp = 9,81N$
Geschwindigkeit	$1 * \frac{m}{s} = 3,6 * \frac{km}{h}$	-
Magnetisches Feld	$1Wb = 1 * Vs$; $1T = 1 * \frac{Vs}{m^2}$	$\mu_0 = 4 * \pi * 10^{-7} \frac{Vs}{Am}$ $\mu_0 = 1,257 * 10^{-6} \frac{Vs}{Am}$
Spule	$1H = 1 * \frac{Vs}{A}$	
Elektrisches Feld	-	
Kondensator	$1F = 1 * \frac{As}{V} = 1 * \frac{s}{\Omega}$	$s_0 = 8,854 * 10^{-12} \frac{As}{Vm}$

10.2 mathematische Formeln

<p>Satz des Pythagoras</p> 	<p>a = Ankathete b = Gegenkathete c = Hypotenuse</p>	$c^2 = a^2 + b^2$
<p>Trigonometrische Funktionen</p> 	<p>a = Ankathete b = Gegenkathete c = Hypotenuse</p>	$\begin{aligned}\sin \varphi &= b/c \\ \cos \varphi &= a/c \\ \tan \varphi &= b/a\end{aligned}$

Anhang